MASTER EN SCIENCES INFORMATIQUES

Génération d'interfaces graphiques pour l'édition d'actions

Penninck, Daniel

Award date:
1999

Faculté Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 1998-1999

Génération d'interfaces graphiques pour l'édition d'actions

Daniel Penninck

Mémoire présenté en vue de l'obtention
du grade de Maître et Licencié en Informatique

Avant-propos

Avant que vous ne commenciez la lecture de ce document, j'aimerais remercier plusieurs personnes sans l'aide de qui ce document n'aurait pu voir le jour :

- mon promoteur Jean Ramaekers sans qui ce stage n'aurait pu avoir lieu, et qui m'a judicieusement guidé tout au long de la rédaction de ce mémoire.
- l'équipe d'*OpenMaster* de la société BULL à Clayes-sous-Bois qui m'a accueilli et intégré en son sein et au contact de qui j'ai énormément appris tant au niveau connaissances informatiques qu'au niveau richesse humaine.

  Je remercie tout particulièrement Dominique Philippi et Christian Ritter pour leur soutien, leur aide précieuse durant le stage et leur conseils avisés lors de l'élaboration de ce document.
- Aimé Kassa pour ses nombreux commentaires et suggestions qui m'ont permis de rendre ce document plus agréable à la lecture .

A tous et toutes, un **grand merci**.

Table des matières

1. Introduction 5

2. Présentation générale d'ISM Configuration Manager 7

2.1. Introduction 7

2.2. Grandes fonctions d'ISM Configuration Manager 8

2.3. Concepts de base d'ISM Configuration Manager 8

2.4. Actions 8

3. La problématique 10

3.1. Introduction 10

3.2. Java 11

3.2.1. L'intérêt du Java 11

3.2.2. Java Bean 13

3.3. L'application OpenMaster 15

3.3.1. Introduction 15

3.3.2. OpenMaster 15

3.3.2.1. Architecture générale 15

3.3.2.2. Architecture avant exécution 17

3.3.3. Le Webtop 18

3.3.4. La sous-application ICM 20

3.4. EXtensible Markup Language (XML) 21

4. Les solutions 26

4.1. Introduction 26

4.2. Générateurs d'interfaces graphiques 26

4.2.1. TaskGuide viewer 26

4.2.2. Prototype 28

4.2.3. Générateur maison 31

4.3. Présentation de différents outils en Java 31

4.3.1. EXtensible Stylesheet Language (XSL) 31

4.3.2. Parser 34

4.3.2.1. Introduction 34

4.3.2.2. Document Object Model (DOM) 35

4.3.2.3. SAX 37

4.4. Ensemble des solutions envisagées 38

4.4.1. Solution 1 38

4.4.2. Solution 2 39

4.4.3. Solution 3 40
4.4.4. Solution 4 40
4.4.5. Solution 5 41
4.4.6. Solution 6 41
4.4.7. Conclusion 42

5. Le prototype réalisé 43

5.1. Analyse fonctionnelle 43
5.1.1. Les fonctionnalités 43
5.1.2. Modélisation d'une action 44
5.1.2.1. Introduction 44
5.1.2.2. L'entête du document 44
5.1.2.3. La description d'une action 44
5.1.2.4. L'aide d'une action 45
5.1.2.5. Les paramètres d'une action 46
5.1.2.6. Les fonctions SML 47
5.1.2.7. La valeur d'un paramètre 48
5.1.2.8. Les modes d'exécution 48
5.2. Analyse technique 50
5.2.1. L'architecture choisie 50
5.2.2. Le moteur graphique "GUI" 51
5.2.2.1. Introduction 51
5.2.2.2. Les méthodes d'ajout de composants 55
5.2.2.3. Les méthodes de manipulation des composants 58
5.2.3. Le moteur "ICM" 60

6. Conclusion 62

7. Bibliographie 64

8. Abréviations 65

9. Annexe 66

9.1. Dtd ICM action 66

9.2. Tags de TaskGuide Viewer 68

9.3. Dtd Prototype 70

1. Introduction

Mon stage s'est déroulé au sein de l'équipe *OpenMaster* de la société BULL, aux Clayes-sous-Bois. Cette équipe s'occupe essentiellement du développement de la plate-forme d'administration *Integrated System Management (ISM)*. OpenMaster est la nouvelle appellation du produit ISM. Le sujet proposé concerne un des composants de cette plate-forme d'administration: ISM *Configuration Manager* (ICM). ICM a pour rôle de faciliter la gestion d'un parc informatique à partir d'un point d'administration centralisé. Il va permettre de gérer la configuration *hardware* aussi bien que les *softwares* installés. ICM est donc une application qui permet de maîtriser un très grand nombre de composants du système d'information. L'une des principales fonctionnalités offerte par ICM est l'exécution d'action sur le parc informatique. Par action, on entend toute opération d'administration telle que l'ajout d'un utilisateur sur toutes les machines NT et Unix du parc informatique.

OpenMaster, comme toutes les applications, évolue dans le temps, et c'est ainsi que se sont succédées les différentes versions de l'application; chaque nouvelle version offrant de nouvelles fonctionnalités. Actuellement, une des futures fonctionnalités que les membres de l'équipe OpenMaster veulent offrir en plus, est la gestion du parc informatique à partir d'un point distant de l'ordinateur d'administration. En d'autres termes, ils veulent permettre à l'administrateur de pouvoir gérer le parc informatique à partir de n'importe quel endroit de la société, en se connectant sur l'ordinateur d'administration avec un simple navigateur Web. Il va donc falloir créer les interfaces utilisateurs qui vont s'afficher dans le navigateur.

Donc, ICM, comme beaucoup d'autres composantes d'ISM, aurait dû créer ses propres interfaces qui allaient s'afficher dans le navigateur. Comme le nombre d'action était élevé et, afin de faciliter l'ajout de nouvelles actions, l'équipe d'ICM décida de générer de façon dynamique les interfaces (en cours d'exécution). L'objectif du stage consistait à modéliser les actions d'ICM et à développer un générateur d'interfaces graphiques permettant l'édition et le déclenchement d'actions. Afin de faciliter l'approche du problème, ce document est divisé en cinq grandes parties :

- La présentation générale d'ISM Configuration Manager : ce chapitre va définir les grandes fonctions, les concepts de base propres à ICM (notamment la définition de ce qu'est une action). La définition de l'action est très importante car les actions sont la pierre angulaire du mémoire.

- La problématique : ce chapitre est le prolongement du chapitre précédent, puisque maintenant que nous savons ce qu'est une action (et les fonctionnalités d'ICM), nous pouvons analyser les besoins et l'environnement de travail. Il exposera exactement en quoi consiste la problématique (ce qui est demandé de faire) et le contexte de travail : nous verrons l'intérêt du Java par rapport à d'autres langages, une présentation sommaire du langage de modélisation *eXtensible Markup Language* (XML), et enfin une présentation de l'application coordinatrice d'ISM dans laquelle notre application devra s'insérer.
- Les solutions envisagées : maintenant que nous connaissons les concepts, la problématique et l'environnement de travail, nous pouvons envisager les différentes solutions possibles. Nous commencerons par une brève présentation et analyse des applications déjà existantes (des générateurs d'interfaces graphiques à partir d'une modélisation XML) et des outils Java (*parser*, ...). Nous finirons par une présentation des différentes possibilités de combinaisons de ces applications entre elles et avec nos propres développements éventuels.
- Le prototype réalisé : au cours de ce chapitre, nous expliquerons la solution choisie et nous présenterons l'application développée qui résout notre problématique. Elle sera le moteur graphique de ICM.
- Conclusion : dans ce chapitre nous verrons les perspectives d'avenir de l'application développée, à savoir, les fonctionnalités non encore implémentées et les extensions possibles.

2. Présentation générale d'ISM Configuration Manager

2.1. INTRODUCTION

ISM Configuration Manager est la solution à la nécessité de maîtriser un très grand nombre de composants du système d'information.

Avec ISM Configuration Manager, maîtriser signifie :

- Maîtriser les configurations matérielles

Pouvoir détecter rapidement dans l'ensemble du parc les systèmes ayant une mauvaise configuration matérielle.

- Maîtriser les configurations systèmes

Pouvoir homogénéiser la configuration d'un ensemble de systèmes, vérifier cette homogénéité, déclencher des actions correctrices sur les systèmes non conformes.

- Maîtriser les logiciels installés

Pouvoir installer des logiciels là où il le faut, sans donner la liste des systèmes à installer mais en décrivant le modèle des systèmes candidats à l'installation.

- Maîtriser la distribution

Pouvoir configurer de façon cohérente les composants coopérant entre eux sur le réseau, grâce à une vue globale et un point d'administration centralisé.

- Maîtriser la diversité

Pouvoir rassembler logiquement les divers composants du parc informatique en groupes homogènes. Contrôler et agir au niveau de ces groupes et non plus au niveau des machines.

- Maîtriser les évolutions de l'administration

Pouvoir récupérer et injecter dans ISM Configuration Manager des scripts d'administration existants. Ces derniers pourront être utilisés sous forme d'actions, ou invoqués, le cas échéant, lors des contrôles de cohérence. ISM est ouvert grâce à la possibilité d'introduire des actions écrites par l'administrateur.

2.2. GRANDES FONCTIONS D'ISM CONFIGURATION MANAGER

Les grandes fonctions d'ISM Configuration Manager sont :

- administration d'un très grand nombre de systèmes en une seule opération à travers le concept de domaine.
- administration des systèmes, basée sur la notion d'état désiré avec le concept de configuration de cohérence.
- maintien de l'intégrité du parc grâce au contrôle de cohérence.
- prise en compte du mode déconnecté. ISM Configuration Manager gère les systèmes connectés en permanence au réseau, mais aussi ceux qui ne le sont que de temps en temps.

2.3. CONCEPTS DE BASE D'ISM CONFIGURATION MANAGER

ISM Configuration Manager est fondé sur quatre concepts : domaine, configuration de référence, contrôle de cohérence, action.

Les concepts de domaine, configuration de référence et contrôle de cohérence sont liés :

- le domaine est un ensemble de systèmes (machines) ayant des caractéristiques communes;

- la configuration de référence est l'ensemble des valeurs nominales de ces caractéristiques communes;

- le contrôle de cohérence compare les valeurs réelles des caractéristiques à leurs valeurs nominales, c'est-à-dire la configuration réelle du domaine à sa configuration de référence.

Seul le concept d'action sera développé dans la suite de ce document.

2.4. ACTIONS

Une action est destinée à être exécutée sur une machine cible, généralement pour accomplir une tâche d'administration. ISM Configuration Manager est livré avec un ensemble d'actions de base pour AIX, Windows NT, HPUX, SunOS, IRIX que l'administrateur peut compléter en ajoutant des actions spécifiques. Dans la suite de ce paragraphe, afin de bien faire comprendre ce qu'est une action, nous allons vous donner des exemples d'actions propres à certaines plates-formes. Par contre, nous développerons ultérieurement les caractéristiques d'une action (les paramètres, le libellé, le mode d'exécution, etc.).

a) Exemple d'actions livrées avec ISM Configuration Manager:

- créer un utilisateur;
- créer un répertoire;
- supprimer un répertoire;
- changer la permission d'accès d'un fichier ou d'un répertoire;
- etc.

b) Exemple d'actions propres à NT, livrées avec ISM Configuration Manager:

- créer un groupe global;
- supprimer un groupe global;
- supprimer un utilisateur d'un groupe global;
- ajouter un membre existant (utilisateur ou groupe global) à un groupe local existant;
- supprimer un membre existant (utilisateur ou groupe global) d'un groupe local existant;
- créer un groupe local;
- supprimer un groupe local;
- etc.

c) Exemple d'actions propres à Unix, livrées avec ISM Configuration Manager:

- monter un système de fichier;
- augmenter un espace de *swap*;
- créer un groupe d'utilisateurs;
- supprimer un groupe d'utilisateurs;
- ajouter une file d'attente pour imprimer sur un système distant;
- afficher la liste des groupes de volume;
- changer le propriétaire d'un répertoire;
- changer le groupe propriétaire d'un répertoire;
- exécuter une commande;
- exécuter un fichier de commande;
- créer un système de fichier et son volume logique;
- etc.

3. La problématique

3.1. INTRODUCTION

Actuellement, l'utilisation d'ISM s'effectue directement à partir de l'ordinateur d'administration sur lequel est installé l'application ISM. Les interfaces utilisateurs sont écrites en SML (System Management Language), un langage propriétaire. L'objectif pour la version suivante d'ISM, baptisée OpenMaster, est de fournir la possibilité d'utiliser l'application ISM à partir d'une machine distante par rapport à l'ordinateur d'administration sur lequel est installé l'application ISM. Pour se faire, il est nécessaire de développer une application cliente qui sert en quelques sorte de boîte de contrôle d'ISM. L'équipe d'OpenMaster décida de développer cette application en Java. ICM comme toutes les autres sous-applications d'ISM doit s'intégrer dans la nouvelle application cliente.

L'intégration d'ICM à l'application cliente va obliger de réécrire les interfaces graphiques de déclenchement des actions. Comme il existe une centaine d'actions, il est impératif de modéliser les actions et de pouvoir générer automatiquement une interface graphique différente pour chaque action. D'autre part, cette modélisation doit pouvoir être assez générale pour supporter de nouvelles évolutions. Une solution à ce problème serait un moteur capable de générer les fenêtres, à partir d'une description de l'action. Le moteur doit pouvoir gérer les objets de type liste de sélections simples, liste de sélections multiples, boîte de dialogues de fichiers, menu, aide, etc. Nous avons choisi le *eXtensible Markup Language* (XML) comme langage de modélisation des fenêtres car il est simple et répandu. XML est un métalangage de définitions de types de documents. XML définit les règles de niveau supérieur (c'est-à-dire la syntaxe) de tout nouveau type de document. Le document XML qui modélise l'action ne devra contenir aucune information de type mise en page mais uniquement la description de l'action. Il faut donc concevoir un moteur capable de transformer de façon automatique et dynamique ces descriptions XML en une interface utilisateur Java. L'utilisation d'une modélisation a pour principal avantage, de permettre à l'application d'avoir une capacité à évoluer en permettant aux utilisateurs autorisés d'ajouter et de modifier des actions sans devoir apprendre le Java puisque les interfaces ne sont pas codées en dur. Néanmoins, chaque modélisation d'action ajoutée devra avoir son pendant dans les fonctions du *framework* (cfr. chapitre 3.3 l'application OpenMaster).

Pour nous, l'interface d'une application est codée en dur si *toutes* les fenêtres de l'application sont connues à l'avance et livrées avec l'application. Dans une

application codée en dur, l'interface n'évolue pas au cours du temps puisqu'elle est programmée et livrée avec l'application. L'ajout d'une nouvelle action dans une application dont l'interface est codée en dur, nécessite de reprendre le code source de l'application, d'y ajouter la nouvelle interface utilisateur et de recompiler le tout. Alors que nous voulons permettre à notre application d'évoluer dans le temps sans pour autant devoir réécrire une seule ligne de code.

Dans la suite de ce chapitre, nous allons passer en revue les avantages de Java puisqu'il s'agit du langage de programmation choisi. Ensuite, nous expliquerons l'architecture de OpenMaster. En effet, notre application ICM n'est qu'une des nombreuses sous-applications d'OpenMaster. Nous verrons comment OpenMaster gère ses sous-applications et comment ICM va s'insérer et interagir avec ce dernier. Ce point est important car il définira le contexte et l'environnement dans lequel notre application devra tourner. Nous finirons ce chapitre par une présentation du langage de modélisation des actions à savoir le XML.

3.2. JAVA

3.2.1. L'intérêt du Java

Java est un langage de programmation orienté objet développé par *Sun Microsystems*. Conçu à partir de C++, Java est simple, concis et portable sur toutes les plates-formes et systèmes d'exploitation. Cette portabilité existe au niveau des sources et des binaires.

Une *applet* est un programme dynamique et interactif qui s'exécute dans une page *web* affichée par un programme de navigation compatible Java. Les *applets* sont de petits programmes créant des animations, des présentations multimédia, des figures, des zones de dialogue avec l'utilisateur, des jeux en temps réel, des jeux réseaux multi-utilisateurs. En fait, les *applets* Java réalisent tout ce qu'un petit programme peut faire. Une *applet* doit être écrite dans le langage Java, compilée avec un compilateur Java et référencée dans les pages *web* HTML. L'installation d'un fichier HTML contenant une *applet* Java sur un site *web* est identique à celle d'un fichier image ou HTML ordinaire. Ainsi, lorsqu'un utilisateur utilise le programme de navigation, il visualise la page et son *applet*. Le programme de navigation la télécharge sur le système local et l'exécute.

De plus, Java est un langage de programmation à part entière. De ce fait, il rend les mêmes services et résout les mêmes problèmes que d'autres langages de programmation comme C ou C++.

Java ne dépend d'aucune plate-forme

Son indépendance vis-à-vis des plates-formes matérielles est son principal avantage. Il est donc tout indiqué pour les applications devant tourner sur différentes plates-formes. Son indépendance se situe à deux niveaux: source et binaire. Au niveau source, les types de données primaires de Java ont la même taille, quelle que soit la plate-forme de développement. Les bibliothèques de classes standard de Java facilitent l'écriture du code qui est ensuite porté de plate-forme en plate-forme sans adaptation. Néanmoins, l'indépendance vis-à-vis de la plate-forme ne s'arrête pas aux fichiers sources. Les fichiers binaires sont également portables, puisqu'ils tournent sur de nombreuses plates-formes sans nécessiter une recompilation, grâce au pseudo-code qui constitue les fichiers binaires de Java (cfr. figure 3.1). Le pseudo-code est un ensemble d'instructions proches du code machine, mais non spécifique à un ordinateur. En réalité, il est spécifique à une machine virtuelle. L'environnement de développement Java comprend un compilateur et un interpréteur. Le compilateur génère le pseudo-code et l'interpréteur permet de l'exécuter. Pour qu'un programme Java soit exécutable sur une plate-forme, il suffit qu'un interpréteur Java ou un navigateur implémentant la machine virtuelle Java soit disponible sur cette plate-forme. L'inconvénient du pseudo-code est sa faible vitesse d'exécution. Les programmes spécifiques à un système, étant directement liés au matériel pour lequel ils sont compilés, tournent plus vite qu'un pseudo-code Java traité par l'interpréteur.

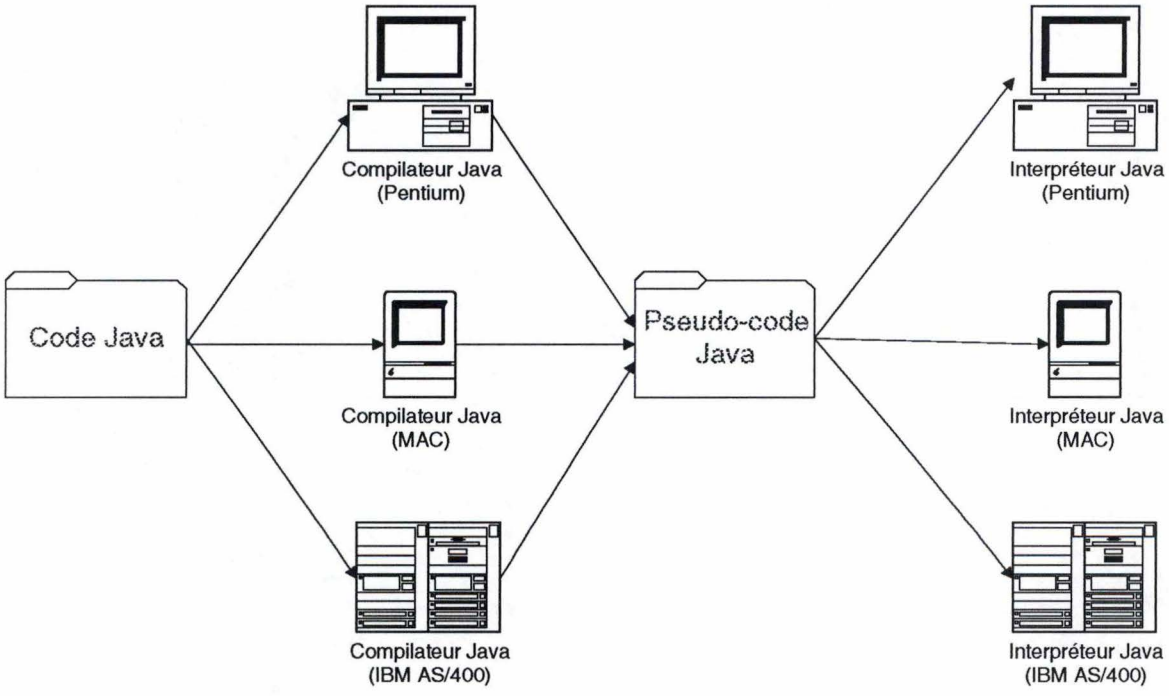

Figure 3.1: l'indépendance de Java vis-à-vis de la plate-forme

Java est orienté objet

Travailler avec un langage et un environnement de programmation orientés objet permet de tirer le meilleur parti de cette méthodologie et de ses capacités. En effet, la création de programmes souples et modulaires ou de codes réutilisables est possible. Java possède des bibliothèques de classes qui fournissent les types de données de base, les possibilités d'entrées/sorties du système (des classes pour la gestion de réseaux, des protocoles Internet) et autres fonctions utilitaires (outils d'interface utilisateur).

Java est facile à apprendre

Java a été conçu pour être concis, simple, facile à écrire, à compiler, à corriger, mais surtout facile à apprendre. Ce langage comporte peu de mots, ce qui augmente sa fiabilité. Ainsi le risque de tomber sur un problème épineux de programmation est réduit. Java reste souple et puissant, malgré sa taille et sa conception simple. De plus, le programmeur ne doit pas se soucier de la récupération, libération de mémoire allouée aux objets car cette dernière est automatique grâce au Garbagge Collector. En effet, c'est ce dernier qui va désallouer les emplacements mémoires occupés par des objets inutiles. Par conséquence, il existe en Java un opérateur *new*() pour créer un objet, mais il n'existe pas d'opérateur *dispose*() de l'objet puisque la gestion est automatique.

3.2.2. Java Bean

En un mot, un *bean* est un composant réutilisable de logiciel qui peut être manipulé visuellement dans un outil de type *builder*. Certains Java *beans* peuvent être de simples éléments GUI comme les boutons. D'autres Java *beans* peuvent être des composants visuels sophistiqués de logiciels tel qu'un *viewer* de base de données. Certains Java *beans* peuvent ne pas avoir d'apparence GUI en soi mais peuvent quand même être manipulés visuellement en utilisant un *builder*. Certains outils de type *builder* peuvent fonctionner complètement de façon visuelle. D'autres *builders* peuvent fournir des facilités d'écriture de classe Java qui interagissent avec et contrôlent un ensemble de *beans*. Chaque Java *bean* diffère des autres en fonction des fonctionnalités qu'il supporte mais ils ont tous les caractéristiques suivantes, propres au Java *bean* :

- support pour l'introspection : ainsi un outil *builder* peut analyser le fonctionnement du *bean*;
- support pour la personnalisation : ainsi un programmeur utilisant un *builder* peut personnaliser l'apparence et le comportement d'un *bean*;
- support pour les événements qui sont utilisés comme moyen simple de communication entre *beans*;

- support pour la manipulation des propriétés du *bean*;
- support pour la persistance : ainsi les caractéristiques d'un *bean* personnalisé dans un *builder* peuvent être sauvegardées et rechargées ultérieurement.

Nous tenons à faire remarquer que, si au départ les *beans* sont prévus pour être manipulés dans un *builder*, ils peuvent tout de même être utilisés via une programmation directe (sans *builder*).

Propriétés, événements et méthodes

Les trois plus importantes caractéristiques d'un *bean* sont ses propriétés dont il permet l'accès, ses méthodes qui peuvent être appelées, et enfin, les événements qu'il peut émettre.

Les propriétés sont des attributs nommés, associés au *bean* et qui peuvent être lus ou écrits en appelant la méthode appropriée du *bean*. Un *bean* peut avoir, par exemple, une propriété "couleur" qui représente la couleur des caractères. Cette propriété peut être lue en appelant la méthode "**get**Couleur()" et modifiée en appelant la méthode "**set**Couleur(C)".

Les méthodes qu'un Java *bean* exporte sont des méthodes Java normales qui peuvent être appelées à partir d'un autre composant ou à partir d'un environnement de script. Par défaut, toutes les méthodes publiques d'un *bean* sont exportées. Néanmoins, un *bean* peut choisir de n'exporter qu'un sous-ensemble de ses méthodes publiques.

Les événements fournissent le moyen à un composant de notifier à d'autres composants que quelque chose d'intéressant est arrivée. En Java, on associe un écouteur d'événements à une source d'événements. Lorsque quelque chose d'intéressant se passe, la méthode appropriée de l'écouteur d'événements est appelée.

3.3. L'APPLICATION OPENMASTER

3.3.1. Introduction

Nous avons vu que Bull a décidé de développer la partie cliente de son application OpenMaster en Java et que ISM Configuration Manager (ICM) est une sous-application d'OpenMaster. ICM peut être vu, d'une certaine manière, comme un module d'OpenMaster. OpenMaster est composé d'un *framework* que l'on contrôle grâce à une application cliente, à savoir le Webtop. Un *framework* sert d'interface entre un client qui requiert un service et les fournisseurs de services. Le *framework* est en quelque sorte un serveur de services. Il offre un environnement de communication pour les clients et les fournisseurs. Le *framework* peut aussi offrir lui-même des services mais il n'est pas forcément le seul fournisseur.

Dans la suite de ce chapitre, nous allons vous présenter les grandes composantes de l'architecture d'OpenMaster. Ensuite, nous expliquerons en quoi consiste le côté client (le Webtop) et enfin, nous situerons ICM dans l'architecture d'OpenMaster.

3.3.2. OpenMaster

3.3.2.1. Architecture générale

OpenMaster est formée de deux grandes composantes: la partie *framework* et la partie *client* (le Webtop). La partie *framework* installée sur une machine distante, est le moteur des opérations que l'on peut effectuer sur l'ensemble des machines. La partie *client* est en quelque sorte la boîte de contrôle du *framework*. Elle va permettre d'afficher les opérations et fonctionnalités possibles; elle va permettre d'ordonner au *framework* d'effectuer certaines opérations et d'avoir un retour informatif sur le déroulement des actions. Le côté client n'est pas uniquement une interface mais est aussi pourvu d'une certaine intelligence. En effet, certaines applications vont communiquer et interagir entre elles au niveau du client. Certaines fonctionnalités ne nécessitent pas toujours un rôle très actif de la part du *framework* et s'exécutent principalement au niveau du client. Le client bénéficie donc d'une certaine autonomie par rapport au *framework*.

L'architecture *framework* étant assez complexe et offrant une multitude de services, nous ne développerons que les services de base nécessaires dans le cadre de notre problématique. Le *framework* comprend deux grandes composantes : le démon HTTP qui va permettre l'accès aux fichiers publics (fichier libre d'accès à toute personne via le réseau interne ou internet) et le SML Wrappers server (SOW).

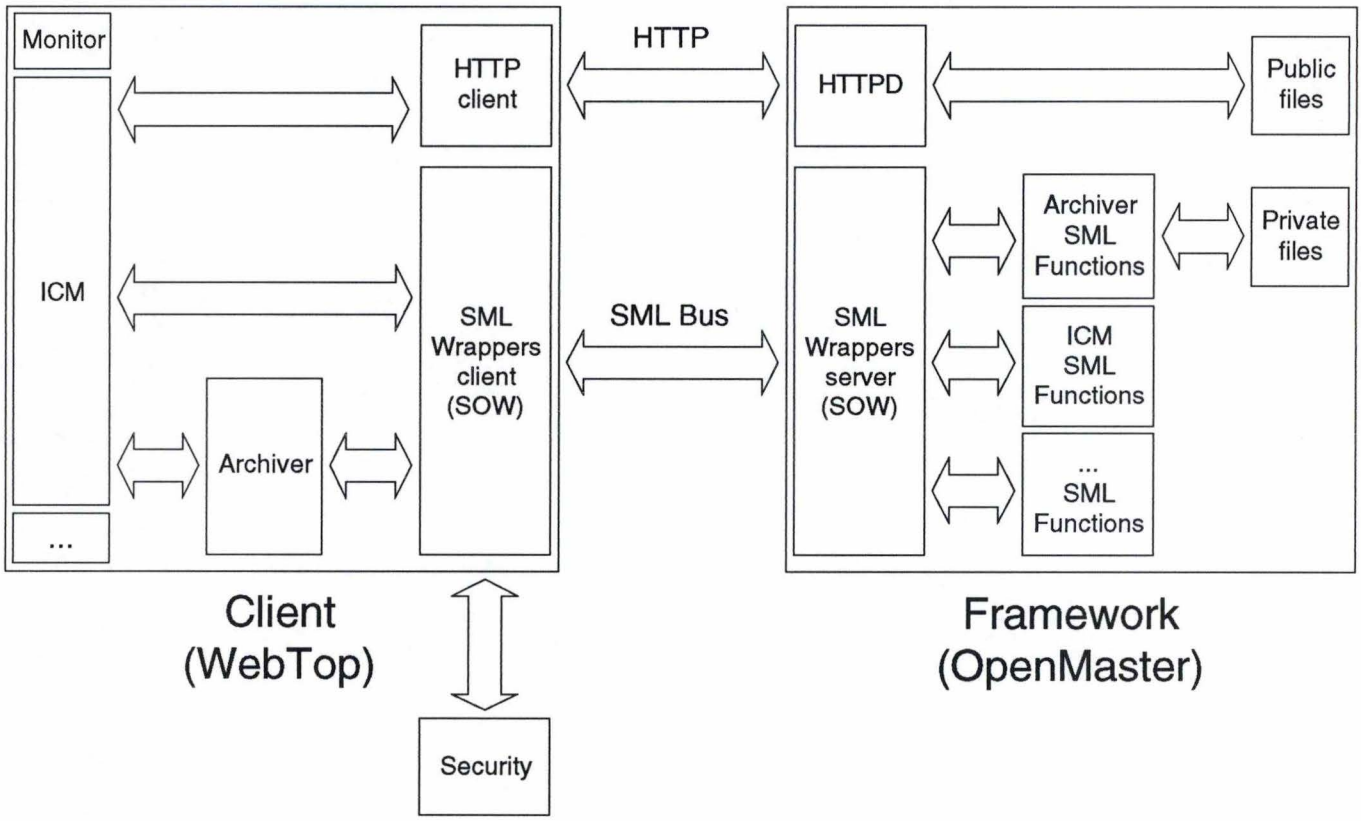

Figure 3.2: l'architecture framework d'OpenMaster

Le SML Wrappers server va permettre de demander l'exécution d'une fonction SML par le *framework* et d'en obtenir un compte rendu, à savoir la bonne ou mauvaise exécution de la fonction et éventuellement les valeurs résultats. Dans le cadre de notre problématique, seuls deux regroupements de fonctions nous intéressent : celui d'ICM et celui de l'*archiver*.

Les fonctions SML de l'*archiver* vont permettre aux composantes du Webtop, non seulement d'accéder et de rapatrier des fichiers du *framework*, mais aussi de les modifier, voire d'en créer de nouveaux. Il est important de remarquer que ces fichiers ne sont pas accessibles via le démon HTTP. Ce sont des fichiers privés qui ne peuvent être accessibles par n'importe qui. Le gestionnaire du réseau est, par exemple, l'une des personnes qui pourrait être amenée à modifier ces fichiers.

Les fonctions SML d'ICM effectuent, réalisent les différentes actions d'ICM (cfr. Exemples précédents). Ce sont donc ces fonctionnalités qui sont le coeur du moteur ICM. Comme nous devions résoudre la problématique de la modélisation et du déclenchement des actions, nous ne développerons pas plus loin, le contenu des fonctions SML propres à ICM. Il existe bien sûr une multitude d'autres fonctions, notamment celles liées aux autres sous-applications, mais celles-ci n'ont aucun intérêt dans le cadre de notre problématique.

Dans cette architecture, nous pouvons remarquer que chaque sous-application d'OpenMaster est divisée en deux parties. La première partie se trouve dans le Webtop et est composée des interfaces utilisateurs ainsi que de certaines

fonctionnalités. La seconde partie est formée d'un ensemble de services disponibles via le *framework* et notamment les fonctions SML propres de la sous-application.

3.3.2.2. Architecture avant exécution

L'architecture présentée dans le point précédent (cfr. figure 3.2) est celle mise en place lors de l'exécution. En réalité, dans l'état initial, le client fait partie intégrante du *framework* (cfr. figure 3.3).

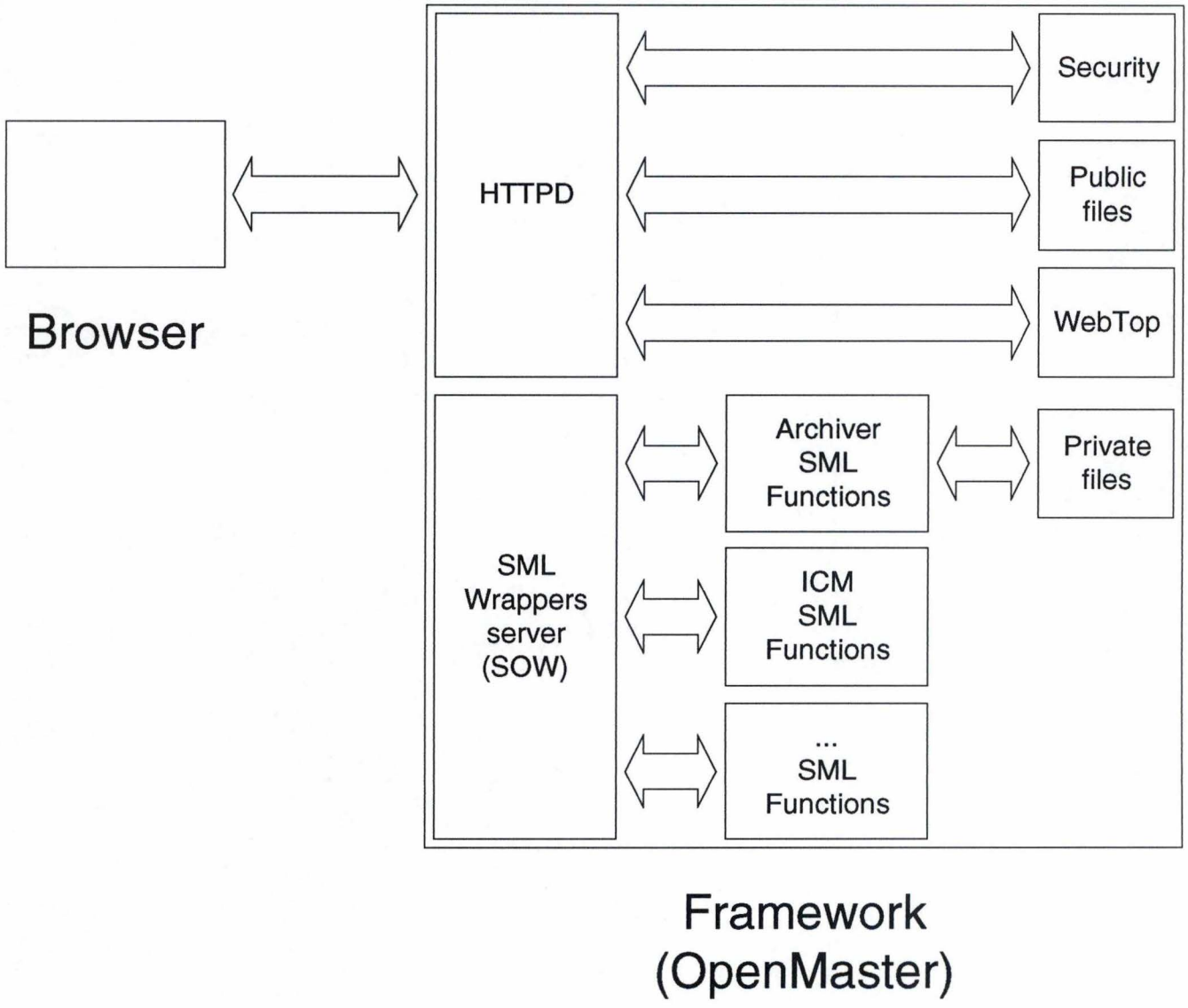

Figure 3.3: l'état initial du framework

Comme le Webtop est entièrement développé sous forme d'*applet* Java, il suffit d'avoir un navigateur pour pouvoir lancer et utiliser l'application ISM. Néanmoins, dans l'état actuel des choses, il faut, avant la première utilisation, aller sur la page publique (HTTP) du *framework* afin d'effectuer une installation des fichiers liés à la sécurité. La sécurité du côté *framework* est constituée, pour l'instant, d'un fichier contenant le nom et le code (généré par le *framework*) de tous les utilisateurs pouvant utiliser le *framework*. D'un autre côté, l'utilisateur doit avoir exécuté l'installation et avoir ainsi son nom d'utilisateur et son mot de passe sur son disque local. Ensuite, lors du lancement et de l'exécution d'ISM, il y aura une vérification de l'existence de l'utilisateur dans le fichier du *framework* et de la correspondance des mots de passe. Une fois l'installation effectuée, il ne reste donc, pour utiliser ISM, plus qu'à charger le Webtop dans le navigateur via le site *web* du *framework*. On se retrouve alors dans l'architecture présentée par la figure 3.2.

3.3.3. Le Webtop

Au sein même du Webtop, les différentes grandes applications d'OpenMaster bénéficient d'une certaine autonomie par rapport au Webtop. En effet, ces dernières sont développées dans des ISMbeans. Un ISMbean est un Java *bean* créé dans le but de permettre au Webtop de gérer l'affichage des applications, les communications entre *beans* et la communication des *beans* avec le *framework*.

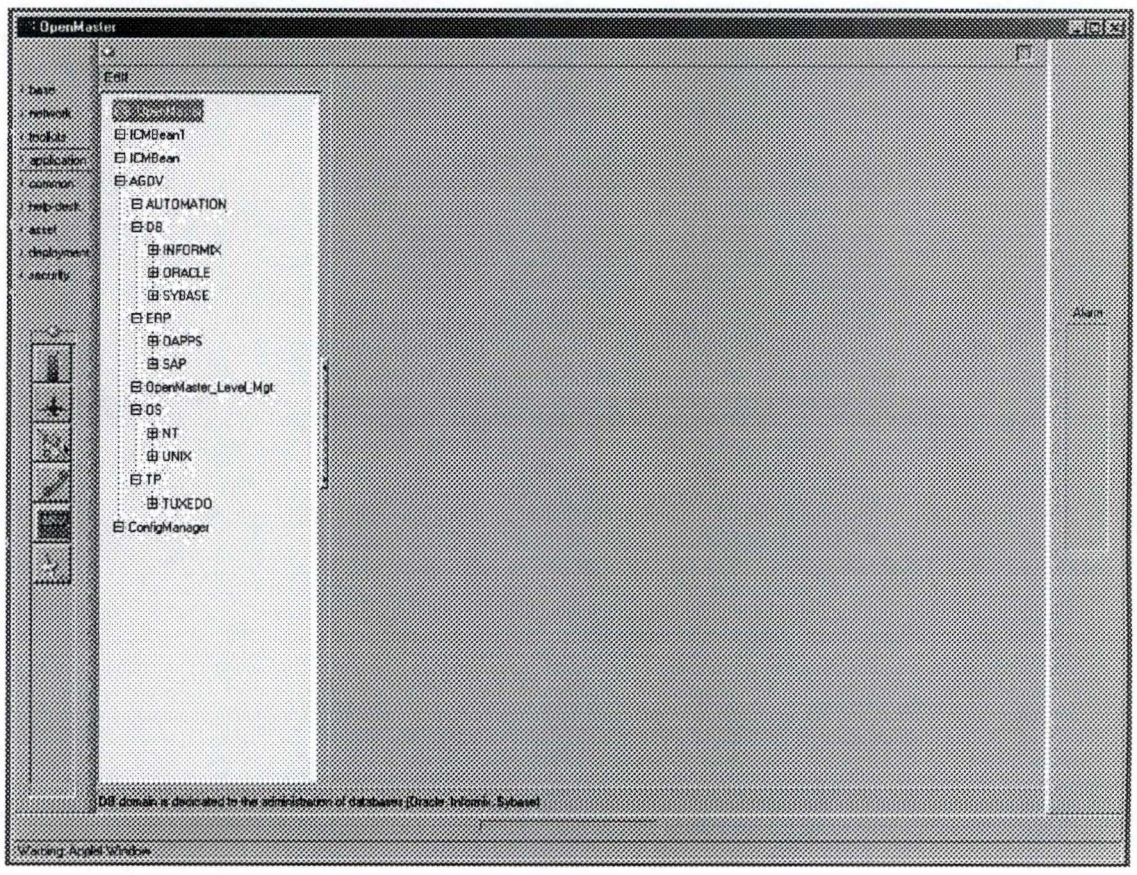

Figure 3.4: le Webtop

Le Webtop est le côté client de l'application OpenMaster au sein duquel les sous-applications vont venir s'intégrer. Le Webtop est une *applet* qui fournit l'environnement de base pour chaque sous-application. Comme on peut le voir sur la firgure 3.4, le Webtop est composé de deux grandes parties : la partie de gauche composée des boutons, de l'arbre (nommé AGOV) et, la partie de droite, composée de l'espace rectangulaire vide dans lequel les sous-applications (de type ISMBean) pourront s'afficher.

Lorsque le bouton ou le noeud de l'arbre représentant une sous-application (ou fonctionnalité) est pressé, le Webtop instancie l'ISMBean associé à cette sous-application (ou fonctionnalité). Un ISMBean est un *bean* constitué d'un *container* auquel on peut ajouter les éléments graphiques. Ce *bean* contient déjà une barre de menu contenant l'*item* de menu "Help". Comme nous l'avons vu, les attributs d'un *bean* sont facilement modifiables et cela permet au Webtop de redimensionner et d'insérer l'ISMBean dans la zone rectangulaire d'affichage de la fenêtre (cfr. figure 3.5).

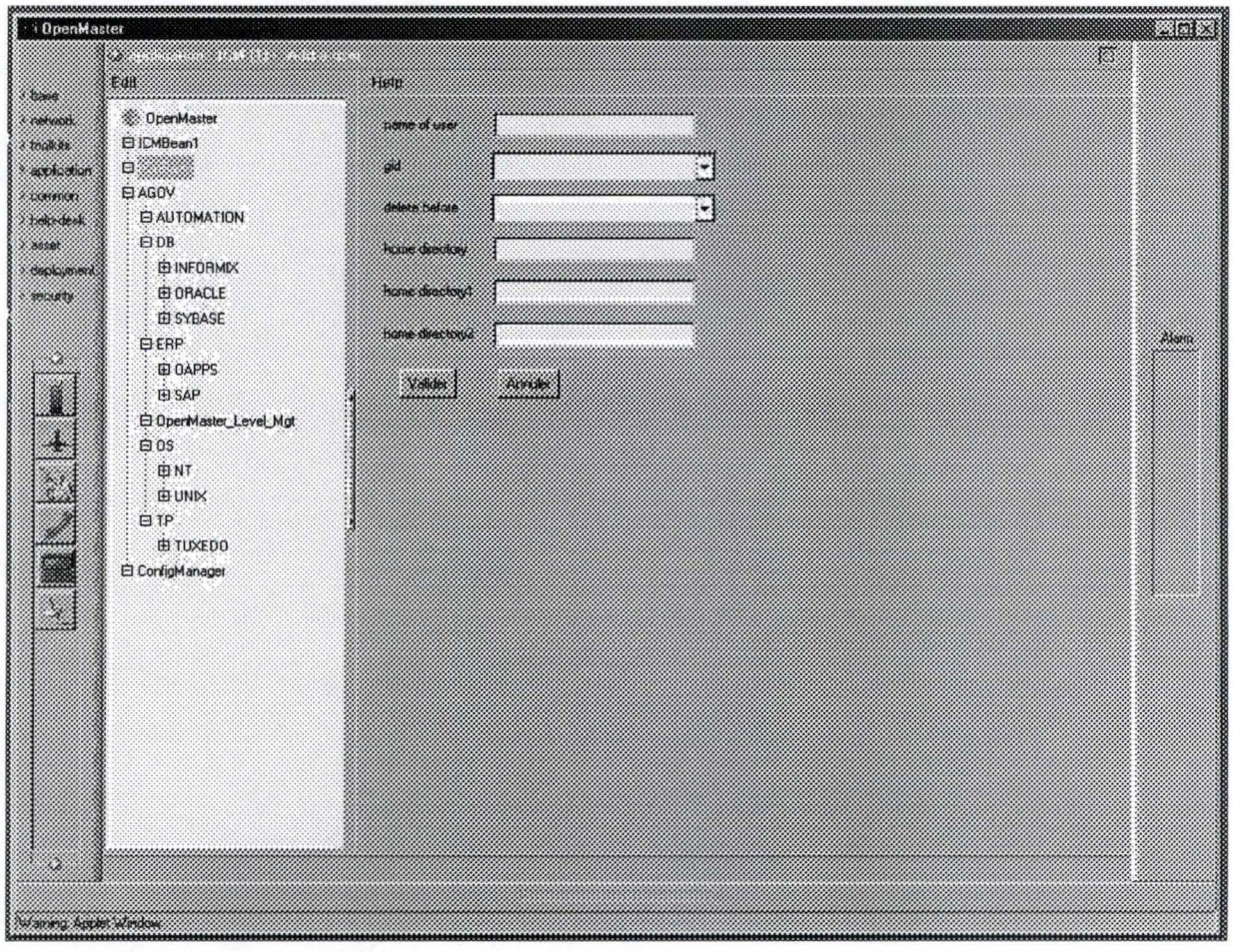

Figure 3.5: la zone d'affichage du Webtop

Le Webtop offre la possibilité d'instancier plusieurs *beans*, mais seul un, sera le *bean* courant (afficher). On peut changer le *bean* courant en utilisant les menus du Webtop. De plus, le Webtop offre la possibilité d'extraire le *bean* de la zone rectangulaire et de l'afficher dans une *frame* (fenêtre indépendante) et vice versa (cfr. figure 3.6).

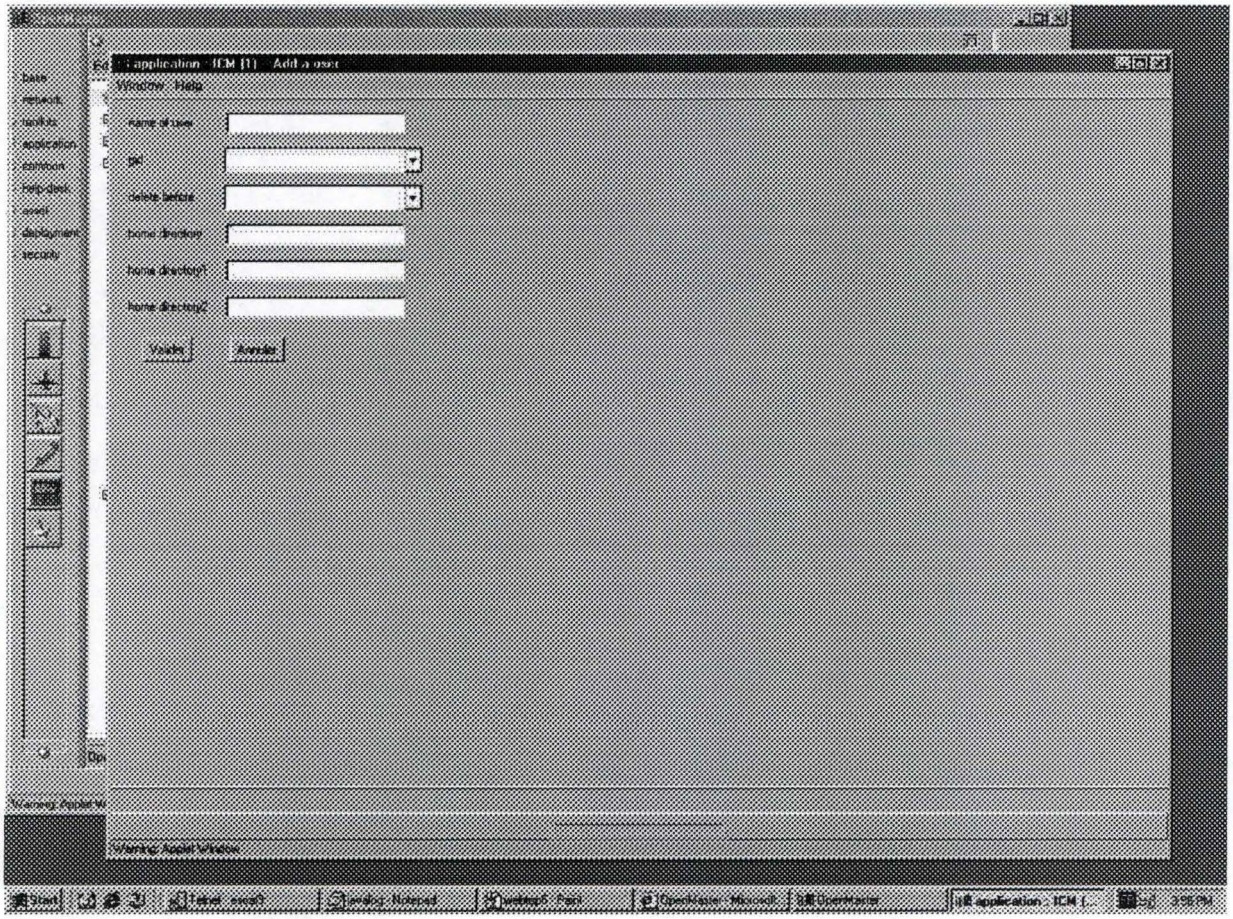

Figure 3.6: l'extraction d'un bean du Webtop

3.3.4. La sous-application ICM

Comme nous vous l'avions expliqué précédemment, ICM est composé de deux grandes parties : un ensemble de fonctions sur le *framework* et une partie application (ICMBean) dans le Webtop. La classe ICMBean est une extension de la classe ISMBean. Lorsqu'un utilisateur clique sur un noeud de l'arbre (dans le Webtop) correspondant à une action d'ICM, le Webtop crée une instance ICMBean et lui transmet certains paramètres dont le nom du fichier modélisant l'action. Cette instance ICMBean devra afficher une fenêtre de saisie des paramètres nécessaires à

l'exécution de l'action ainsi qu'offrir la possibilité de sauvegarder (et reprendre) des jeux de paramètres. Ensuite, il devra offrir la possibilité de déclencher l'action. L'instance ICMBean sera donc amenée à utiliser l'*archiver* pour obtenir les fichiers XML de modélisation des actions, pour obtenir les fichiers *properties*, pour effectuer la sauvegarde des jeux de paramètres d'une action, etc. L'instance ICMBean utilisera aussi le SML Wrappers client pour lancer l'exécution des actions. Les fonctionnalités offertent par l'ICMBean ainsi que son architecture interne seront expliquées dans un chapitre ultérieur.

3.4. EXTENSIBLE MARKUP LANGUAGE (XML)

XML est un format de données pour l'échange de documents structurés sur le *web*. XML est devenu une recommandation du W3C depuis le 18 février 1998 .

Par définition, XML est extensible : aussitôt définie, une extension est universellement utilisable. XML autorise les développeurs à définir un nombre illimité de *tags* pour décrire n'importe quel type d'éléments au sein d'un document.

Alors qu'HTML décrit la présentation de l'information (style, taille, ...), XML décrit les données.

Un exemple :

<ORDER>
<SOLD-TO>
<PERSON>
<LASTNAME> *Layman*</LASTNAME>
<FIRSTNAME> *Andrew*</FIRSTNAME>
</PERSON>
</SOLD-TO>
<SOLD-ON>*19970317* </SOLD-ON>
<ITEM>
<PRICE> *5.95*</PRICE>
<BOOK>
<TITLE> *XML is Beautiful* !</TITLE>
<AUTHOR> *Bos Bert* </AUTHOR>
</BOOK>

</ITEM>

</ORDER>

Les éléments peuvent ainsi être organisés en hiérarchie d'informations; la contre-partie est qu'on ne peut pas placer un *tag* à n'importe quel endroit du document.

Pour qu'un document soit déclaré « valide », chaque nouveau *tag* doit être inclus dans une DTD.

La DTD spécifie la structure logique du document : elle définit les éléments d'une page et leurs attributs ainsi que les relations entre ces éléments (exemple : l'élément « Title » ne peut apparaître qu'au sein de l'élément « Book »).

Si les *tags* ne sont pas corrects ou imbriqués proprement, le *parser* déclare le document XML invalide.

Cependant, pour pallier aux problèmes de performance liés à la vérification de la validité d'un document, un serveur peut très bien offrir un document XML sans sa DTD : les DTDs ne sont pas obligatoires. Dans ce cas, les *parsers* XML peuvent déclarer le document « well-formed » sans avoir à référer à aucune DTD (pas de contrôle sur la définition ou la cohérence des *tags*). Ceci implique quelques contraintes supplémentaires au regard de ce que permet par exemple HTML:

- tous les *start-tags* doivent avoir leurs pendants ("*end-tags*") présents (sauf pour les éléments EMPTY dont le *start-tag* doit se finir par "/"; ex: <BR/>);
- toutes les valeurs d'attributs doivent être mises entre "".

On peut imaginer un document XML comme représentant la linéarisation d'une structure de type arbre. Cet arbre a un certain nombre de nœuds dont nous ne décrirons que le premier :

- les Eléments;
- le Document;
- les « Processing Instruction » (PI) : représentés sous la forme *< ?i ?>*, ils autorisent les documents à contenir des instructions pour des applications;
- les Commentaires : représentés sous la forme *< !—c -- >*, ils permettent d'inclure des notes explicatives destinées à améliorer la compréhension;
- les Data : comme tous les autres nœuds possèdent des délimiteurs '<' et '>', les données n'en ont pas besoin. Tout ce qui n'est pas entre '<' et '>' représente des données.

Les PI, commentaires et data, représentent tous trois des feuilles de l'arbre.

Les Eléments

Déclaration des Eléments dans la DTD

< !ELEMENT anthology (poem+)>

< !ELEMENT poem (title ?, stanza+)>

< !ELEMENT title (#PCDATA)>

< !ELEMENT stanza (line+)>

< !ELEMENT line (#PCDATA)>

La première partie de la déclaration donne le « generic identifier » de l'élément déclaré (exemple : *poem, title*, ..).

La deuxième partie, entre parenthèses, est le « content model » (ou « element's content »), et spécifie quelles occurrences peuvent être contenues dans le corps de l'élément. Ce contenu peut désigner d'autres éléments ou utiliser des mots réservés tels que #PCDATA (« parsed character data ») qui signifie que l'élément défini peut contenir n'importe quel caractère valide.

Les indicateurs d'occurrence indiquent combien de fois l'élément nommé dans le « content model » peut apparaître lors de l'écriture du document :

Signes	Significations
+	Une ou plusieurs occurrences
?	Zéro ou une occurrence
*	Zéro, une ou plusieurs occurrences

Tableau 3.1: les indicateurs d'occurrences

Les connecteurs de groupe indiquent dans quel ordre les éléments définis dans l' « element's content » peuvent apparaître :

Caractères	Significations
,	les composants doivent apparaître dans l'ordre indiqué
&	les composants doivent apparaître mais dans un ordre indifférent
|	un seul des composants qu'il connecte peut apparaître

Tableau 3.2: les connecteurs de groupe

Exemple d'utilisation des éléments précédemment définis dans un document:

<anthology>

<poem><title>The SICK ROSE</title>

<stanza>

<line>O Rose thou art sick.</line>

<line>The invisible worm,</line>

<line>That flies in the night</line>

<line>In the howling storm:</line>

</stanza>

<stanza><line> ...</line><line>...</line> ...

</stanza>

</poem> <!-- more poems go here -->

</anthology>

Attributs

Indépendamment de l' « element's content », les attributs vont permettre de mieux décrire une occurrence d'un élément donné. Les valeurs d'attributs sont toujours représentées dans un document comme une paire « attribut-valeur » à l'intérieur du start-tag d'un élément.

Exemple :

<poem id="P1" status="draft"> ... </poem>

L'élément <poem> a été défini avec deux attributs, id et status, que le processeur XML peut traiter comme il le souhaite (exemple : imprimer un document différemment suivant qu'il est à l'état « draft » ou « revised »).

Exemple de déclaration d'attribut :

< !ATTLIST poem

id ID #IMPLIED

status (draft|revised|published) « draft »>

Le type de l'attribut peut être :

- soit un mot-clé prédéfini comme

  CDATA : la valeur d'attribut peut contenir n'importe quel caractère valide;

  ID : identifiant unique;

  IDREF : pointeur vers un autre élément.

- soit la liste des valeurs possibles de l'attribut (*draft|revised|published*).

La dernière partie de la déclaration spécifie comment le *parser* doit interpréter l'absence de l'attribut concerné :

- soit en fournissant une valeur par défaut;
- soit en utilisant un des mots-clés suivants :

  #REQUIRED : une valeur doit être spécifiée;

  #IMPLIED : la valeur n'est pas obligatoire;

  #CURRENT : si aucune valeur n'est fournie pour une occurrence, c'est la dernière valeur spécifiée qui sera reprise.

4. Les solutions

4.1. INTRODUCTION

Dans ce chapitre, nous allons vous présenter, en premier lieu, des générateurs d'interfaces à partir de fichiers XML ainsi que leurs limites. La présentation sera relativement sommaire puisque nous nous contenterons de donner un exemple d'utilisation pour chaque générateur. Vu la solution adoptée, une analyse détaillée des DTD de ces générateurs n'apporterait pas grand chose dans le cadre de ce mémoire. Néanmoins, vous trouverez les-dites DTD en annexe. Ensuite, nous allons présenter des applications qui pourraient être combinées avec un générateur d'interfaces ou être intégrées dans une solution maison. Finalement, nous présenterons l'ensemble des solutions envisagées à partir de l'enchaînement des différentes applications.

4.2. GENERATEURS D'INTERFACES GRAPHIQUES

Un générateur d'interfaces graphiques est une application capable de construire et d'afficher une fenêtre utilisateur à partir d'une description de cette dernière. Cette description de fenêtre peut être, par exemple, un fichier de format XML. Les applications présentées dans la suite de ce chapitre sont développées en Java.

4.2.1. TaskGuide viewer

Introduction

IBM TaskGuide est une application Java qui permet de construire un «wizard» à partir d'un fichier texte XML. TaskGuide permet de générer une succession d'interfaces utilisateurs (GUI) à partir du fichier XML.

TaskGuide Viewer permet de définir et d'afficher:

- une fenêtre de dialogue d'aide;
- des hyperliens;
- des cases à cocher, des cases d'options, des listes déroulantes, ... ;
- des boutons avec un label spécifié et un état (actif, inactif, caché);

- des libellés, des panneaux, ...

Exemple

Contenu du fichier Xml

```
<sguide>
 <title>Sample Wizard  </title>
 <panel name=main>
  <title>&lt;option&gt; Example</title>
  <p>What's your favorite color? </p>
  <select name=favorite_color>
   <option>Puce </option>
   <option>Magenta </option>
   <option>Chartreuse </option>
   <option fill-in selected> Other: </option>
  </select>
 </panel>
</sguide>
```

Fenêtre générée par TaskGuide

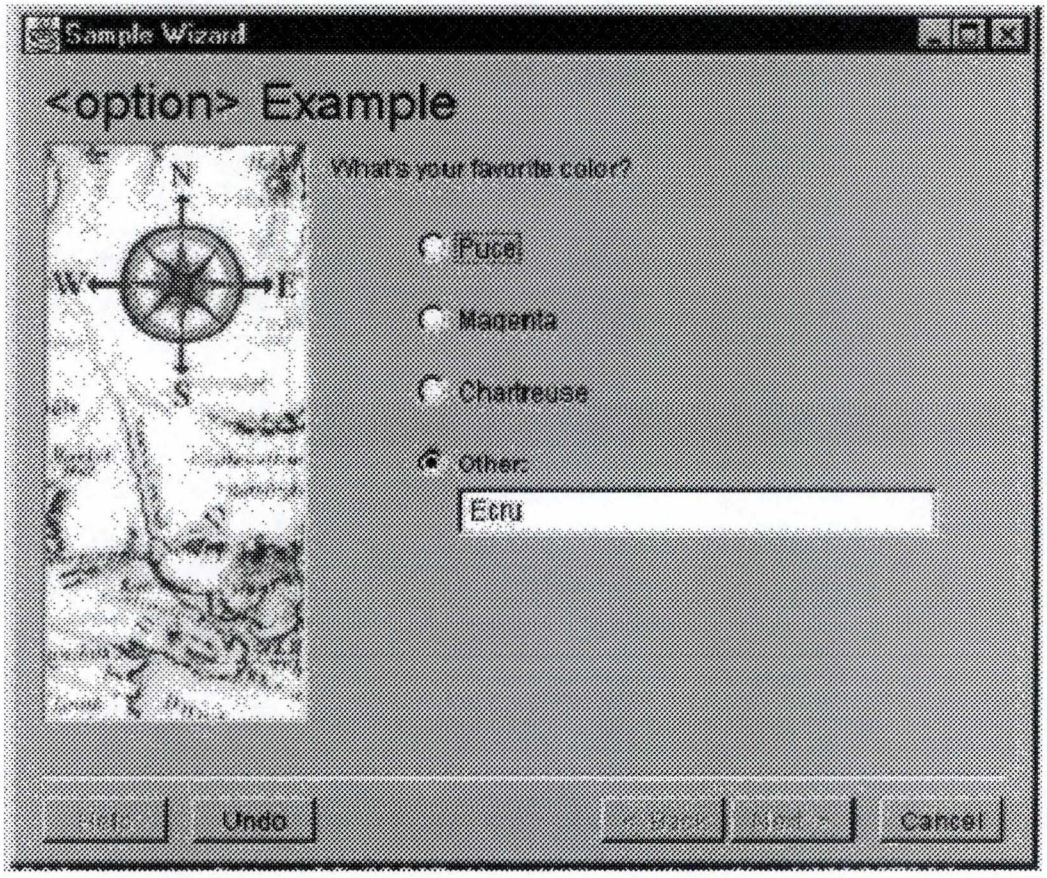

Figure 4.1: TaskGuide viewer

Conclusion

La génération de *wizards* à partir d'une description XML est relativement facile. En effet, le vocabulaire (le nombre de *tags*) est relativement restreint par rapport au générateur Prototype présenté dans le point suivant, et il n'est pas nécessaire de définir la position des différents composants de la fenêtre. Taskguide Viewer est donc plus orienté sur la tâche que sur la mise en page. Taskguide Viewer occupe à peu près 650k, ce qui est raisonnable par rapport à Prototype. La taille de l'application a de l'importance car cette dernière doit être chargée par le réseau et exécutée dans un navigateur.

Par contre, nous tenons à signaler que le code source (fichiers *.java) n'est pas livré avec l'application et donc il est impossible de l'adapter à nos besoins, à notre environnement. Il faudra donc l'utiliser tel quel. De plus, Taskguide Viewer accepte des fichiers XML dont la syntaxe n'est pas rigoureuse.

4.2.2. Prototype

Introduction

Prototype est un programme qui lit un fichier texte et lui donne l'apparence d'une application ("produces the look and feel of an application"). La structure du fichier texte doit être du XML valide. Prototype lit cette description structurée d'un programme et génère l'interface utilisateur de ce dernier.

Exemple

L'exemple proposé crée le prototype d'un magasin. Il est constitué de deux panneaux. Le panneau de gauche montrera un arbre des départements et les produits à vendre dans chacun de ces départements. Le panneau de droite changera en fonction de l'article choisi sur le panneau de gauche. La description de l'arbre est contenue dans le fichier "store.xml". Le panneau de droite est décrit par le fichier "mpants.xml".

Contenu du fichier "store.xml"

```
<?xml version="1.0" ?>
<!DOCTYPE Window SYSTEM "proto.dtd">
<Window Title="General Store" Top="15" Left="15" Width="400" Height="250">
  <Panel ID="MainPanel" Background="white">
    <FrameSet Percent="40">
      <Panel ID="TreePanel">
        <Tree Pos="2,2,150,200" Caption="Store">
          <TreeFolder Caption="Men">
```

```
          <TreeLeaf Caption="Pants">
           <onMouse Event="Clicked" Target="ProductPanel" Language="XML">
            <script>mpants.xml
            </script>
           </onMouse>
          </TreeLeaf>
          <TreeLeaf Caption="Shirts"></TreeLeaf>
         </TreeFolder>
         <TreeFolder Caption="Women">
          <TreeLeaf Caption="Dresses"></TreeLeaf>
          <TreeLeaf Caption="Shoes"></TreeLeaf>
         </TreeFolder>
        </Tree>
       </Panel>
       <Panel ID="ProductPanel">
       </Panel>
      </FrameSet>
      </Panel>
     </Window>
```

Contenu du fichier "mpants.xml"

```
<?xml version="1.0" ?>
<Panel>
 <Label Caption="Style:" Pos="4,4,100,19" Justification="right"></Label>
 <ComboBox Pos="110,4,80,19" Value="Sport|Golf"
   Editable="no"></ComboBox>
 <Label Caption="Waist:" Pos="4,31,100,19" Justification="right"></Label>
 <ComboBox Pos="110,31,40,19" Value=" |28|30|32|34|36|38|40|42|44|46|48|50|52"
   Selected="32" Editable="no"></ComboBox>
 <Label Caption="Price US$:" Pos="4,58,100,19" Justification="right"></Label>
 <ComboBox Pos="110,58,100,19"
   Value=" |19.99-24.99|25.00-29.99|30.00-34.99|35.00-39.99|40.00-49.99"
   Selected="30.00-34.99" Editable="no"></ComboBox>
 <Button id="bt1" Pos="115,105,100,23" Caption="Search">
  <onMouse Event="Clicked" Target="Cont" Language="XML">
   <script>
    <![CDATA[<Grid Pos="10,10,270,80">
     <GridColumn Caption="Style"></GridColumn>
     <GridColumn Caption="Waist"></GridColumn>
     <GridColumn Caption="Price"></GridColumn>
     <GridData Value="Sport|32|22.00,Sport|32|34.00"></GridData>
    </Grid>]]>
   </script>
  </onMouse>
 </Button>
```

```
 <Container ID="Cont" Pos="10,150,290,180">
  <etchedborder></etchedborder>
 </Container>
</Panel>
```

Fenêtre générée par Prototype

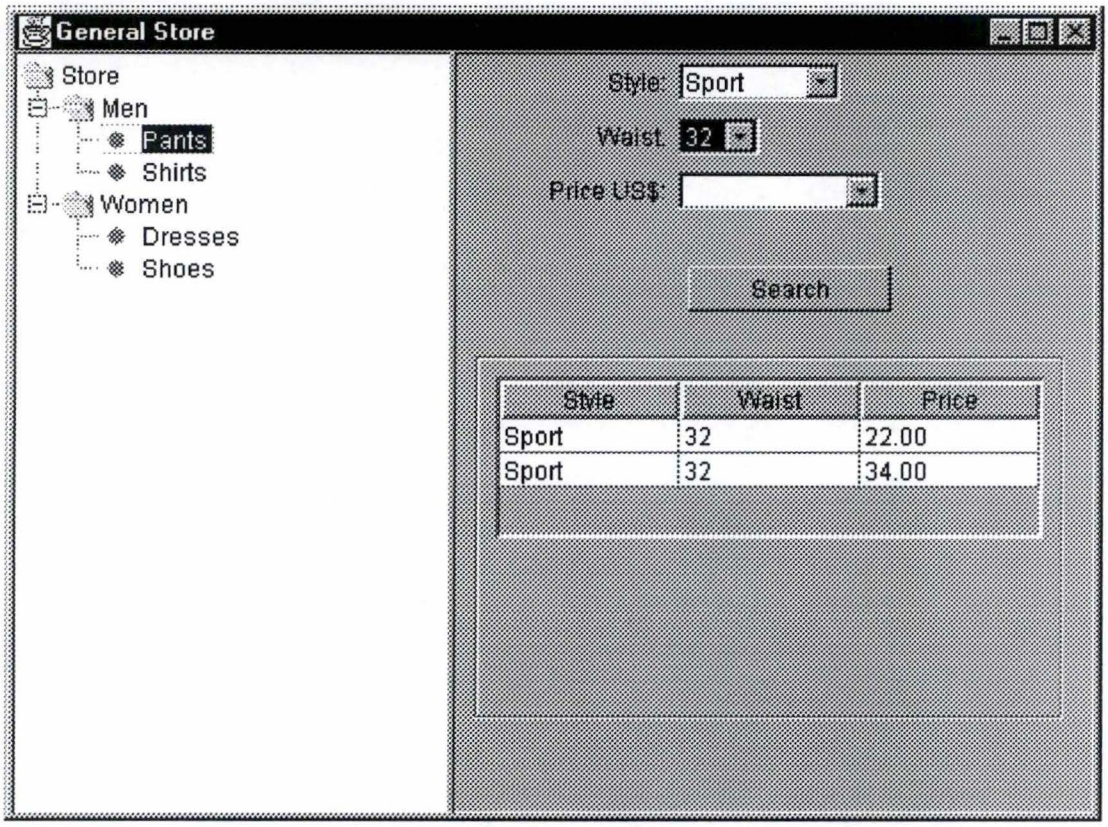

Figure 4.2: Prototype

Conclusion

Prototype a une approche beaucoup plus mise en page que Taskguide Viewer. En effet, il faut définir la position exacte des différents composants de la fenêtre. Cette approche lui permet de générer des interfaces équivalentes à celle codées en dur. Il pourrait, donc, être vu comme un outil de *serialization* (la sauvegarde d'une fenêtre sous forme de fichier XML) d'une interface. Son vocabulaire très étendu et son approche mise en page le rendent plus générique que Taskguide Viewer.

Néanmoins, cette même richesse de vocabulaire et son approche mise en page rendent l'application relativement plus difficile à utiliser que Taskguide Viewer et

surtout plus lente. A nouveau, les fichiers sources ne sont pas livrés avec l'application.

4.2.3. Générateur maison

Une autre possibilité consiste à créer notre propre moteur de génération de fenêtres. Ce moteur utiliserait les classes (*toolkit*) développées par d'autres équipes telles que Bwt de KLG Package (boutons, *labels*, etc.), ISM Map Toolkit (Koala), etc. Un tel moteur se baserait sur un *parser* pour analyser le fichier de description de la fenêtre. Notre moteur devrait être capable de créer la fenêtre, de l'afficher et de gérer les événements possibles sur cette fenêtre. Dans ce cas-ci, nous envisageons de développer un moteur spécifique à la génération de fenêtres d'**action** ICM alors que les outils tels que Prototype et Taskguide Viewer, eux, permettent de générer n'importe quelle fenêtre. Un générateur spécifique nous permettrait d'avoir une DTD plus simple et ainsi, faciliter la tâche du gestionnaire qui devrait ajouter de nouvelles actions. De plus, un générateur spécifique permettrait un certain gain de place et de performance. En effet, il ne faut pas oublier que notre application devra se charger dans un navigateur via le réseau internet ou local. De plus, développer notre propre application permet d'avoir une totale maîtrise du code. Nous pouvons l'adapter et le faire évoluer selon nos besoins. Nous n'aurions sans doute pas le même niveau de liberté avec une application clé en main.

4.3. PRESENTATION DE DIFFERENTS OUTILS EN JAVA

4.3.1. EXtensible Stylesheet Language (XSL)

Introduction

XSL est à l'origine destiné à générer des *slides*. En effet, XSL habille le code XML source de descriptions d'affichage. Ce processus est similaire au CSS pour les pages HTML.

XSL est un langage d'expression de feuilles de style. Il est composé de deux parties :

- un langage pour transformer les documents XML;
- un vocabulaire XML pour spécifier la sémantique de la mise en page ("formatting semantics").

Une feuille de style XSL spécifie la présentation d'une classe de documents XML en décrivant comment une instance de la classe est transformée en un document XML qui utilise le vocabulaire de mise en page ("formatting vocabulary").

Exemple

Voici une feuille de style simple :

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl"
                xmlns:fo="http://www.w3.org/TR/WD-xsl/FO"
                result-ns="fo"
                indent-result="yes">

  <xsl:template match='/'>
    <fo:page-sequence font-family="serif">
      <fo:simple-page-master name='scrolling'/>
      <fo:queue queue-name='body'>
        <xsl:process-children/>
      </fo:queue>
    </fo:page-sequence>
  </xsl:template>
  <xsl:template match="title">
    <fo:block font-weight="bold">
      <xsl:process-children/>
    </fo:block>
  </xsl:template>

  <xsl:template match="p">
    <fo:block>
      <xsl:process-children/>
    </fo:block>
  </xsl:template>

  <xsl:template match="emph">
    <fo:sequence font-style="italic">
      <xsl:process-children/>
    </fo:sequence>
  </xsl:template>
</xsl:stylesheet>
```

Voici un document source XML :

```
<doc>
  <title>An example</title>
  <p>This is a test.</p>
  <p>This is <emph>another</emph> test.</p>
</doc>
```

Le résultat de l'application de la feuille de style au document source :

```
<fo:page-sequence xmlns:fo="http://www.w3.org/TR/WD-xsl/FO"
 font-family="serif">
  <fo:simple-page-master name="scrolling"/>
  <fo:queue queue-name="body">
    <fo:block font-weight="bold">An example</fo:block>
    <fo:block>This is a test.</fo:block>
    <fo:block>This is
      <fo:sequence font-style="italic">another</fo:sequence>
    test.</fo:block>
  </fo:queue>
</fo:page-sequence>
```

Conclusion

XSL est très orienté sur la mise en forme de documents. Il permet de transformer un fichier XML en un fichier XML enrichi d'informations de mise en page. XSL propose un vocabulaire de mise en page ("formatting semantics"). Ensuite, on peut combiner le fichier XML enrichi de la mise en page avec un moteur (par exemple un générateur de page HTML, ...).

Il est relativement difficile de combiner XSL avec un générateur d'interfaces n'utilisant pas exclusivement le vocabulaire de mise en page ("formatting semantics") d'XSL: le fait, par exemple, qu'XSL ne permet pas de gérer directement des entiers (et les opérations arithmétiques) pose un problème pour générer un fichier XML directement compatible avec la DTD de Prototype. En effet, pour ce moteur, il faut préciser la position de chaque élément au pixel près : il faut donc calculer la position d'un élément par rapport à l'élément précédent. Ce que XSL ne permet pas de faire.

Nous tenons à terminer en signalant que pendant le stage (et actuellement), XSL et les moteurs compatibles avec son vocabulaire sémantique n'étaient que des versions béta.

4.3.2. Parser

4.3.2.1. Introduction

Un *parser* est un programme qui se charge de faire une analyse syntaxique sur une source respectant un certain format (une grammaire). Il fait appel à un analyseur lexical qui lui renvoie les *tokens* (les mots acceptables du langage) de cette source. Le *parser* vérifie que les *tokens* renvoyés par l'analyseur lexical sont conformes à une certaine grammaire. Etant donné un ensemble de *tokens* acceptables et une grammaire, ils sont capables de générer un programme Java permettant d'analyser des fichiers respectant cette grammaire.

On distingue deux types majeurs d'API XML :

- Une « tree-based API» qui compile un document XML en une structure interne de type arbre et qui permet aux applications de parcourir cet arbre. DOM est un standard pour les « tree-based APIs »;
- Une « event-based API » qui reporte les événements de *parsing* (telle que la rencontre du début et la fin d'un élément) directement à l'application via des *callbacks*, et qui ne construit pas, d'habitude, d'arbre interne. L'application doit implémenter les méthodes qui vont traiter les différents événements. SAX est un standard pour les *parsers* XML de type « event-based API ».

Avec les *parsers* respectant le standard SAX, il est possible de générer un arbre DOM en utilisant certaines classes intermédiaires. Ces dernières traitent les événements rencontrés lors du *parsing* et construisent l'arbre DOM.

Exemples de parser XML :

- Microstar Aelfred;
- Microsoft's MSXML;
- IBM' s XML for Java;
- James Clark's XP;
- Tim Bray's Lark.

Description du parser Aelfred

Aelfred est un parser XML particulièrement convenable pour être utilisé dans les applets Java.

En effet, voici quelques-uns de ses avantages qui nous intéressent:

- Aelfred est le plus petit possible afin de ne pas augmenter trop le temps de chargement de l'*applet* : il fait 26K au total (15 K dans un fichier compressé de format jar);
- Aelfred utilise le moins de classes possibles afin de minimiser le nombre de connexions HTTP nécessaires pour les *applets* : il est composé de deux classes;
- Aelfred est compatible avec la plupart des implémentations Java et des plates-formes;
- Aelfred possède un temps d'exécution très faible.

4.3.2.2. Document Object Model (DOM)

Cette spécification définit une interface indépendante de toute plate-forme et de tout langage qui permet aux programmes et aux scriptes d'accéder et de mettre à jour de façon dynamique le contenu, la structure, le style de documents. DOM fournit un ensemble standard d'objets pour représenter les documents HTML et XML, un modèle standard sur la manière de combiner ces objets, et une interface standard pour les accéder et les manipuler.

Exemple table :

```
<TABLE>
  <TBODY>
    <TR>
      <TD>Shady Grove</TD>
      <TD>Aeolian</TD>
    </TR>
    <TR>
      <TD>Over the River, Charlie</TD>
      <TD>Dorian</TD>
    </TR>
  </TBODY>
</TABLE>
```

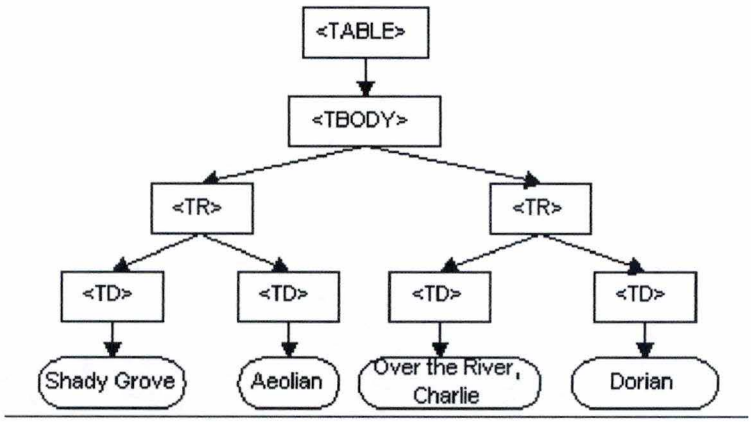

Figure 4.3: représentation DOM de l'exemple table

L'arbre est chapeauté par une classe Java Document. Cette dernière possède une variable qui pointe sur le premier noeud de l'arbre et les méthodes qui permettent de créer de nouveaux noeuds (commentaire, CDATA, texte, ...).

Chaque noeud implémente l'interface Node. Elle possède donc une variable qui indique le type de noeud (cfr. tableau ci-dessous), une variable qui indique le noeud parent, une variable qui indique tous les noeuds fils, une variable qui pointe sur le premier noeud fils, une variable qui pointe sur le dernier des fils, une variable qui pointe directement sur le noeud précédent, une variable qui pointe sur le noeud suivant, une variable qui indique le nom du noeud (cfr. tableau 4.1), une variable qui indique la valeur du noeud (cfr. tableau 4.1), une classe qui indique les attributs XML dans le cas d'un noeud de type *element*.

Types de noeuds	**Nom du noeud**	**Valeur du noeud**
Element	nom du tag	Null
Attr	nom de l'attribut	Valeur de l'attribut
Text	#text	Contenu du noeud texte
CDATASection	#cdata-section	Contenu de la section CDATA
Entity	Nom de l'entité	Null
Comment	#comment	Contenu du commentaire

Tableau 4.1: caractéristiques d'un noeud DOM

Chaque noeud possède aussi des méthodes qui permettent d'insérer un nouveau noeud avant un noeud existant, de remplacer un noeud fils par un autre noeud, d'enlever un noeud fils, d'ajouter un noeud, de dupliquer un noeud.

4.3.2.3. SAX

Comme évoqué précédemment, SAX est un standard pour les *parsers* XML de type « event-based API ». Il définit le format des différentes classes (ainsi que leur méthode), des interfaces que doivent utiliser les *parsers* pour être compatibles SAX. Il définit aussi la manière dont une application va faire appel aux *parsers* et la manière dont elle va obtenir les informations (XML et erreurs). Le but premier est d'uniformiser la manière d'utiliser les *parsers* et ainsi de les rendre interchangeables, sans devoir adapter les applications qui les exploitent.

Afin de mieux comprendre comment fonctionne une « event-based API », reprenons l'exemple suivant:

```
<TABLE>
  <TBODY>
    <TR>
      <TD>Shady Grove</TD>
       <TD>Aeolian</TD>
    </TR>
    <TR>
      <TD>Over the River, Charlie</TD>
      <TD>Dorian</TD>
    </TR>
  </TBODY>
</TABLE>
```

Une interface « event-based API » coupe la structure du document en une série d'événements linéaires:

```
start document
start element: TABLE
start element: TBODY
start element: TR
start element: TD
characters: Shady Grove
end element: TD
start element: TD
characters: Aeolian
end element: TD
end element: TR
start element: TR
start element: TD
characters: Over the River, Charlie
end element: TD
start element: TD
characters: Dorian
end element: TD
end element: TR
```

end element: TBODY
end element: TABLE
end document

A chaque type d'événement correspond une méthode de traitement qui sera appelée lorsque l'événement se produit. Cette méthode est développée par l'utilisateur et lui permet de traiter l'événement à sa manière et avec ses propres structures de données. Comme nous réagissons au fur et à mesure que les événements se produisent, nous n'avons pas besoin de stocker le contenu du fichier XML en mémoire. Nous pouvons donc traiter d'énormes fichiers malgré la contrainte limitative de la disponibilité de la mémoire.

4.4. ENSEMBLE DES SOLUTIONS ENVISAGEES

Nous allons maintenant combiner les différents outils et ainsi générer un ensemble de solutions qui résolvent la problématique. Pour chaque solution, nous donnerons les avantages et les inconvénients.

4.4.1. Solution 1

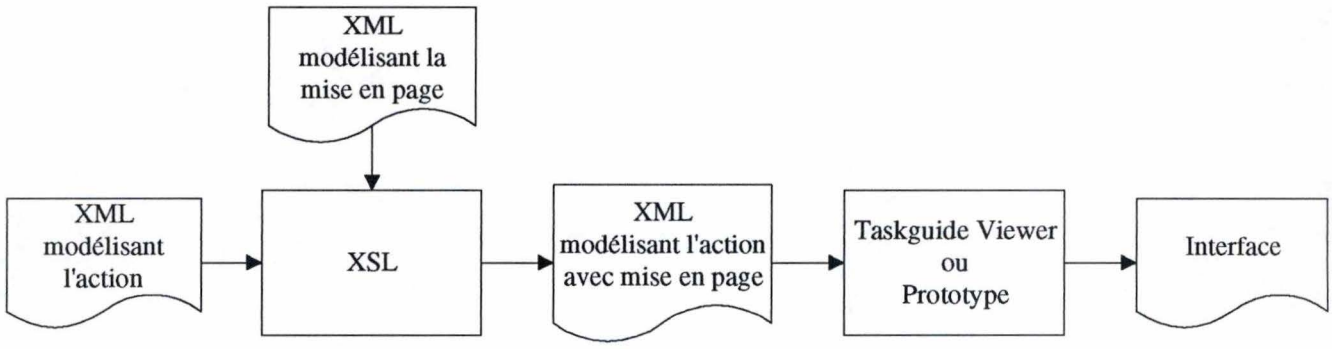

Figure 4.4: solution 1

- Avantage: permet une mise en page qui n'est pas codée en dur puisqu'elle est définie par un document XML ("formatting vocabulary").
- Inconvénients: nous enchaînons deux moteurs (applications) ce qui est relativement lourd (lent). De plus, certaines incompatibilités directes entre les entrées et sorties des moteurs paraissent difficilement résolvables. XSL est une version béta dont le comportement n'est pas toujours correct. La solution n'est pas directement compatible avec l'environnement de la problématique. Par exemple, l'interface utilisateur générée n'est pas contenue dans un ISMBean.

4.4.2. Solution 2

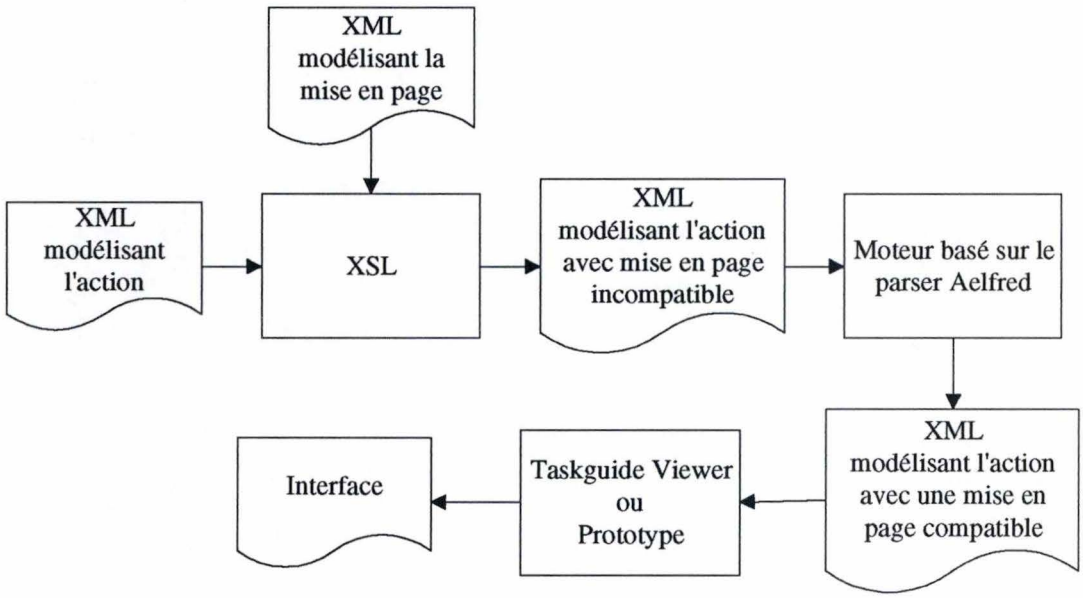

Figure 4.5: solution 2

- Avantages: permet une mise en page qui n'est pas codée en dur puisqu'elle est définie par un document XML ("formatting vocabulary"). De plus, par rapport à la solution 1, les entrées et sorties des moteurs sont directement compatibles.
- Inconvénients: nous enchaînons trois moteurs, ce qui est relativement très lourd (lent). En plus, la solution n'est pas directement compatible avec l'environnement de la problématique. XSL est une version béta dont le comportement n'est pas toujours correct.

4.4.3. Solution 3

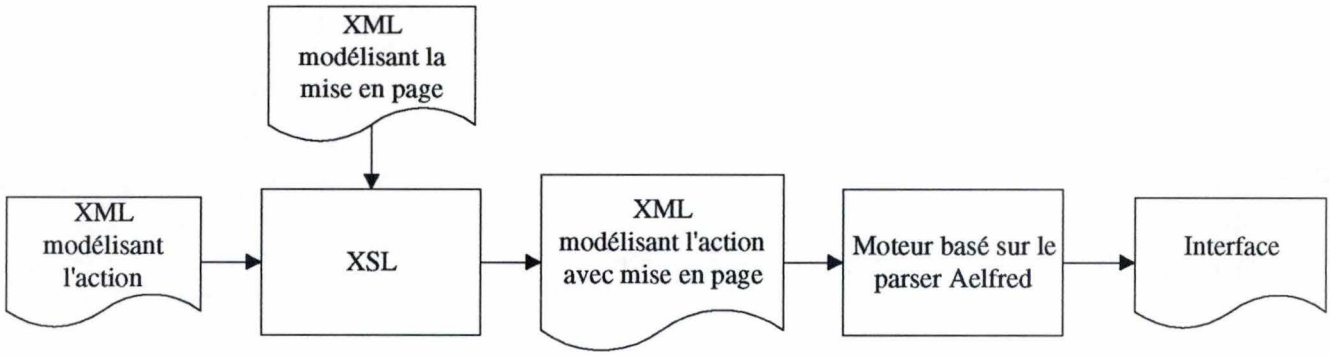

Figure 4.6: solution 3

- Avantages: permet une mise en page qui n'est pas codée en dur puisqu'elle est définie par un document XML ("formatting vocabulary"). En plus, elle est directement compatible avec l'environnement de la problématique.
- Inconvénients: nous enchaînons deux moteurs ce qui est relativement lourd (lent). De plus, XSL est une version béta dont le comportement n'est pas toujours correct.

4.4.4. Solution 4

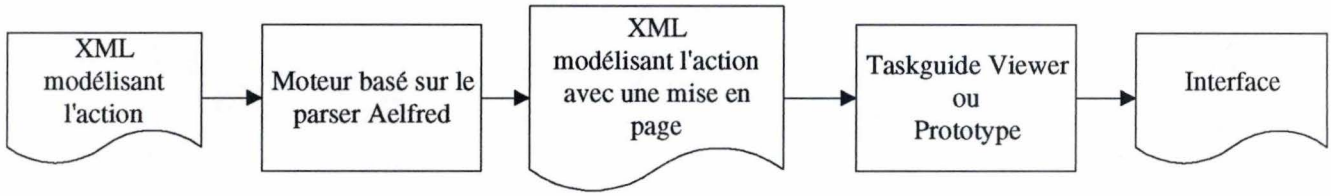

Figure 4.7: solution 4

- Avantage: elle n'utilise que des applications pleinement testées (plus de béta).
- Inconvénients: nous enchaînons deux moteurs ce qui est relativement lourd (lent). De plus, la mise en page est codée en dur dans le moteur basé sur le *parser* Aelfred puisqu'il n'y a pas de document XML modélisant la mise en page.

La solution n'est pas directement compatible avec l'environnement de la problématique.

4.4.5. Solution 5

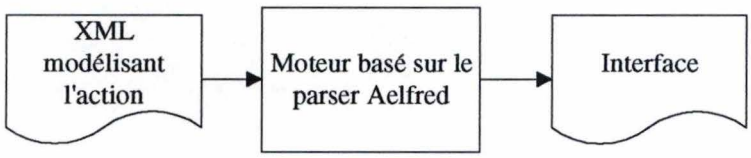

Figure 4.8: solution 5

- Avantages: elle n'utilise que des applications pleinement testées (plus de béta). De plus, elle est relativement légère (et rapide) puisqu'il n'y a plus qu'un seul moteur spécialisé dans le problème. Elle est directement compatible avec l'environnement de la problématique.
- Inconvénient: la mise en page est codée en dur.

4.4.6. Solution 6

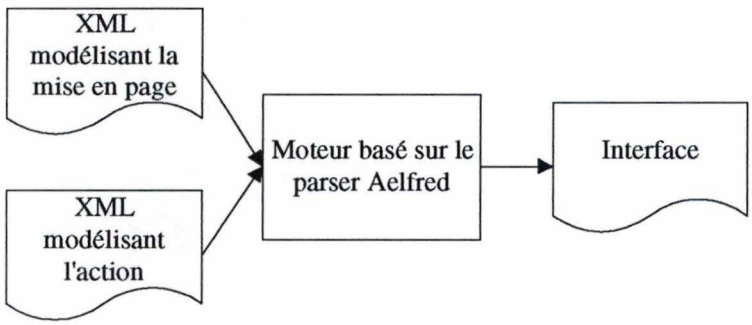

Figure 4.9: solution 6

- Avantages: elle n'utilise que des applications pleinement testées (plus de béta). De plus, elle est relativement légère (et rapide) puisqu'il n'y a plus qu'un seul moteur spécialisé dans le problème. Elle est directement compatible avec l'environnement de la problématique. La mise en page n'est pas codée en dur.

- Inconvénient: c'est la solution qui nécessite le plus de développement et de temps de mise en place.

4.4.7. Conclusion

Comme nous l'avons vu dans ce chapitre, il existe de nombreuses solutions. Ces solutions sont elles-mêmes basées sur l'enchaînement de plusieurs applications. Chacune de ces solutions a ses avantages et ses inconvénients. Pourtant, on ne peut pas vraiment dire qu'une solution est meilleure que les autres. Le choix d'une solution doit tenir compte de plusieurs critères :

- le temps de développement et de mise en place: utiliser une solution basée uniquement sur des applications déjà existantes, permet une mise en place rapide. Tandis que le développement d'une solution entièrement maison nécessite un investissement en temps et en hommes;
- la rapidité de la solution finale : il ne faut pas oublier que la solution mise en place doit être la plus rapide possible puisque la génération de l'interface utilisateur du déclenchement d'une action n'est qu'une partie du processus. En effet, l'exécution de l'action en elle-même nécessite un certain temps;
- les ressources utilisées: ICM n'est pas la seule application qui devra tourner dans le Webtop, ce qui implique que nous devons utiliser un minimum de ressource espace mémoire et CPU. Nous ne devons pas oublier non plus, que les classes Java nécessaires à la mise en place de la solution, devront occuper le moins de place possible car elles devront être chargées, via le réseau, en même temps que le Webtop.

Comme il est difficile d'évaluer les différents critères pour les solutions de type maison (moteur basé sur la *parser* Aelfred), l'équipe d'ICM a décidé de développer un prototype de la solution 5 afin de mieux se rendre compte des performances de cette dernière, des difficultés de mise en place, de la maîtrise technique nécessaire.

5. Le prototype réalisé

5.1. ANALYSE FONCTIONNELLE

5.1.1. Les fonctionnalités

Dans ce chapitre, nous allons présenter les fonctionnalités que devra avoir notre solution:

- La fonction principale de notre application est la génération à partir d'une modélisation XML stockée sur le *framework* de l'interface d'édition de l'action et la gestion du déclenchement de l'exécution de l'action. Par interface d'édition, on entend, une fenêtre dans laquelle on permet à l'utilisateur d'entrer les paramètres nécessaires à l'exécution de l'action et de choisir le type d'exécution.
- Une autre fonctionnalité liée à la première est de permettre à l'utilisateur de sauver les valeurs introduites ou choisies dans un fichier. Cette sauvegarde est appelée jeu de paramètres. Certains jeux de paramètres typiques seront fournis à l'utilisateur avec l'application. Naturellement, s'il y a sauvegarde d'un jeu de paramètres, il faut aussi permettre de pouvoir recharger un jeu de paramètres.
- La modélisation et la gestion de l'interface, doivent pouvoir tenir compte d'un environnement multi-lingues. En effet, l'interface de l'action devra être générée dans la langue du navigateur.
- A chaque action est associée une aide que l'utilisateur devra pouvoir consulter lors de l'édition de l'action.
- Les valeurs de certains paramètres proposés à l'utilisateur ne sont pas toujours connues à l'avance (dans la modélisation) mais sont parfois le résultat de l'exécution d'une fonction du *framework*.
- Il existe quatre modes d'exécution possibles (constraintCheck, execute, execCheck, rollBack) qui seront expliqués dans la modélisation de l'action.

5.1.2. Modélisation d'une action

5.1.2.1. Introduction

Pour résoudre la problématique, nous avons besoin d'une modélisation des actions. Cette dernière, comme vous le savez maintenant, est développée en XML. Nous voulions que la modélisation de l'action soit la plus simple possible et qu'elle ne contienne que la description de l'action et aucune information de type mise en page. Nous allons donc développer ici la DTD (cfr. Chapitre sur XML) qui va définir la manière de construire un fichier de description d'actions. Comme nous l'avons expliqué, l'application doit pouvoir sauvegarder et reprendre des jeux de paramètres. Un jeu de paramètres est composé de l'ensemble des valeurs de paramètres introduites par l'utilisateur. Après réflexion, nous avons décidé que la modélisation en XML d'une action pouvait aussi servir à la sauvegarde des jeux de paramètres. En effet, il suffit de sauvegarder la modélisation de l'action et de lui associer les valeurs des paramètres.

5.1.2.2. L'entête du document

Nous avons décidé qu'un document de description d'actions doit commencer par l'ouverture d'un *tag* ICM. Ce *tag* est en quelque sorte le père de tous les autres sous-*tags* et permet de déclarer, identifier le fichier comme appartenant à ICM. Dans la problématique telle qu'elle nous avait été exposée, on nous a laissé entrevoir comme future fonctionnalité, la possibilité d'envisager l'enchaînement de plusieurs actions. Il en résulte que le *tag* ICM permet l'occurrence de deux éléments: la modélisation d'une action ou la modélisation d'une liste d'actions.

Voici la déclaration XML:

<!ELEMENT icm (action|listAction)>

Comme dans la problématique, il n'était pas initialement question de traiter un enchaînement d'actions, nous ne les avons pas formellement modélisées mais nous avons tout de même laissé la porte ouverte dans notre DTD.

Voici la déclaration XML:

<!ELEMENT listAction (#PCDATA)>

5.1.2.3. La description d'une action

Une action doit posséder un identifiant (idAction) et un type (type). De plus, une action peut posséder un libellé mais ce dernier peut être exprimé de deux manières:

soit on donne directement le libellé (label), soit on donne l'identifiant du libellé dans le fichier *properties* (propertiesLabel). En effet, tout texte, devant apparaître dans une GUI, doit de préférence être dans la langue choisie par l'utilisateur. Donc, plutôt que de donner le libellé tel quel, on doit pouvoir donner son identifiant dans les fichiers *properties* (un fichier *properties* par langue). Lors de la génération de l'interface de l'action, on affichera le texte associé à l'identifiant dans le fichier de la langue choisie. Une action est composée d'une occurrence d'éléments représentant l'aide de l'action, la liste des paramètres, des options d'exécution et éventuellement d'une description du jeu de paramètres.

Voici la déclaration XML:

```
<!ELEMENT action
(listParameterSetDescription?,help,parameterList,optionExecution)>

<!ATTLIST action
 idAction          ID      #REQUIRED
 type              CDATA   #REQUIRED
 propertiesLabel   ID      #IMPLIED
 label             CDATA   #IMPLIED
>
```

Dans le cas où le fichier XML modélise une sauvegarde d'un jeu de paramètres, il faut prévoir un *tag* afin de sauvegarder un texte libre de descriptions du jeu de paramètres. Ce texte pourra apparaître dans les boites de dialogues de chargement de jeux de paramètres. Ce texte a été donné par l'utilisateur au moment de la sauvegarde.

Voici la déclaration XML:

```
<!ELEMENT listParameterSetDescription (#PCDATA)>
```

5.1.2.4. L'aide d'une action

Une action possède une aide associée afin de guider l'utilisateur. Cette aide aussi peut être, soit donné tel quel, soit être dans les fichiers *properties* et être référencée par un identifiant. Ici, le texte libre contenu entre l'ouverture et la fermeture du tag représente directement le texte d'aide ou l'identifiant. En effet, il n'était pas très ergonomique d'intégrer l'aide en tant qu'attribut du *tag* puisque la valeur de cette dernière risquait d'être très longue. Nous avons donc ajouté un attribut (isProperties) permettant de savoir si le texte libre entre l'ouverture et la fermeture du *tag* était l'aide en elle-même ou son identifiant.

Voici la déclaration XML:

```
<!ELEMENT help (#PCDATA)>
```

<!ATTLIST help

isProperties (%boolean;) #REQUIRED

>

Dans la modélisation de l'aide, nous avons utilisé le type bouléen qui n'existe pas en tant que tel en XML. Il s'agit de l'utilisation d'une entité qui est définie comme suit:

<!ENTITY % boolean "(true|false)">

5.1.2.5. Les paramètres d'une action

Comme nous l'avons déjà dit, une action peut avoir zéro à plusieurs paramètres. Ces paramètres sont regroupés entre l'ouverture et la fermeture du *tag* "parameterList".

Voici la déclaration XML:

<!ELEMENT parameterList (parameter)*>

Chaque paramètre d'une action possède un identifiant ainsi qu'un libellé. De nouveau, le libellé peut, soit être directement donné ("label"), soit être référencé dans les fichiers *properties* ("propertiesLabel"). Comme certaines actions peuvent avoir des paramètres facultatifs, nous avons dû introduire un attribut ("mandatory") afin de savoir si le paramètre est facultatif ou obligatoire. Chaque paramètre est caractérisé par un type. En effet, un paramètre peut être un champ texte à remplir par l'utilisateur, une sélection d'un choix parmi une liste, une sélection de plusieurs choix parmi une liste, un texte libre introduit par l'utilisateur. Afin de modéliser les types possibles de paramètres, nous avons introduit la nouvelle entité qui suit:

<!ENTITY % typeParameter
"TEXTFIELD|SIMPLECHOICE|MULTICHOICE|TEXTAREA">

Un paramètre est composé éventuellement d'une valeur par défaut ("defaultValue"), d'une liste de valeurs, d'une référence à une fonction, de la valeur du paramètre lors de la sauvegarde du jeu de paramètres. Les différents sous-*tags* envisageables seront développés ci-dessous.

Voici la déclaration XML:

<!ELEMENT parameter (defaultValue?,(listValue|function)?,parameterSet?)>

<!ATTLIST parameter

idParameter	ID	#REQUIRED
propertiesLabel	ID	#IMPLIED
label	CDATA	#IMPLIED

```
type            (%typeParameter;)     #REQUIRED
mandatory       (%boolean;)           #REQUIRED>
```

Certains paramètres nécessitent une liste de valeurs; c'est notamment le cas avec les paramètres de type simple ou multi-choix. Cette liste de valeurs est utilisée pour proposer les choix possibles.

Voici la déclaration XML:

```
<!ELEMENT listValue (value)+>
```

Certains paramètres possèdent une valeur, une ou plusieurs sélections par défaut.

Voici la déclaration XML:

```
<!ELEMENT defaultValue (value)+>
```

Dans le cas de la sauvegarde d'un jeu paramètre, la valeur courante du paramètre sera sauvegardée dans le tag parameterSet.

Voici la déclaration XML:

```
<!ELEMENT parameterSet (value)+>
```

5.1.2.6. Les fonctions SML

Certaines fonctions du *framework* peuvent être utilisées pour calculer les valeurs proposées dans certains paramètres et pour le déclenchement de l'action. Une fonction possède un nom qui lui sert d'identifiant sur le *framework*. Elle peut posséder un à plusieurs paramètres.

Voici la déclaration XML:

```
<!ELEMENT function    (parameterFunction)*>
<!ATTLIST function
 name                 ID                  #REQUIRED
>
```

Un paramètre d'une fonction peut avoir deux formes: soit on donne directement ou par référence la ou les valeurs du paramètre de la fonction, soit on donne l'identifiant d'un autre paramètre de l'*action*. Dans ce dernier cas, la valeur du paramètre de la fonction est donc celle choisie ou introduite par l'utilisateur dans un autre paramètre de l'action.

Voici la déclaration XML:

```
<!ELEMENT parameterFunction (value)+>
<!ATTLIST parameterFunction
 isIdParameter      (%boolean;)     #REQUIRED >
```

5.1.2.7. La valeur d'un paramètre

Le *tag* "value" est utilisé pour modéliser deux choses bien distinctes: dans le premier cas, il modélise la ou les valeurs d'un paramètre de l'action, et dans l'autre cas, il modélise la ou les valeurs d'un paramètre d'une fonction (cfr. supra). Dans le cas d'un paramètre de l'action, il sert à modéliser les valeurs par défaut, les valeurs possibles dans une liste à sélections et éventuellement la valeur du paramètre au moment de la sauvegarde du jeu de paramètres. Une valeur peut à nouveau être, soit exprimée directement ou bien être référencée dans un fichier *properties* ("isProperties"). Il faut aussi remarquer que la valeur affichée à l'écran (directement ou par référence) et sélectionnée par l'utilisateur est une valeur exprimée de façon compréhensible pour l'utilisateur mais ne correspond pas forcément à l'expression de la valeur attendue par les fonctions du *framework*. Il faut donc introduire un attribut qui contiendra la valeur retour à passer aux fonctions SML si la valeur affichée associée est sélectionnée ("returnValue").

Voici la déclaration XML:

```
<!ELEMENT value      (#PCDATA)>
<!ATTLIST value
 isProperties       (%boolean;)     #REQUIRED
 returnValue        CDATA           #IMPLIED
>
```

5.1.2.8. Les modes d'exécution

Il existe quatre modes d'exécution: l'exécution en elle-même, la vérification des contraintes, l'annulation d'une action, et la vérification d'une bonne exécution.

Voici la déclaration XML:

```
<!ELEMENT optionExecution
(constraintCheck?,execute?,execCheck?,rollBack?)>
```

Le mode d'exécution "constraintCheck" permet de vérifier que toutes les conditions préalables à l'exécution sont bien remplies. Ce mode permet donc de s'assurer qu'une exécution ultérieure se passera convenablement.

Voici la déclaration XML:

<!ELEMENT constraintCheck (function)>

Le mode d'exécution "execute" permet l'exécution pure et simple de l'action.

Voici la déclaration XML:

<!ELEMENT execute (function)>

Le mode d'exécution "execCheck" permet de vérifier que l'exécution d'une action s'est bien déroulée.

Voici la déclaration XML:

<!ELEMENT execCheck (function)>

Le mode d'exécution "rollBack" permet de défaire les conséquences de l'exécution d'une action. Il s'agit en quelque sorte de la fonction retour en arrière.

Voici la déclaration XML:

<!ELEMENT rollBack (function)>

5.2. ANALYSE TECHNIQUE

5.2.1. L'architecture choisie

Finalement, nous avons décidé de développer notre propre moteur graphique. Notre moteur graphique peut-être divisé en deux grandes parties: la première va générer une structure DOM et la seconde va l'analyser pour générer l'interface. Le premier module (cfr. figure 5.1) reçoit en entrée, grâce à l'*archiver*, le fichier XML modélisant l'action à afficher et à exécuter ensuite. Le *parser*, compatible SAX, analyse le fichier XML et interagit avec les classes DOM chaque fois qu'un *token* est identifié. A la fin du processus, l'arbre DOM modélisant l'action est généré.

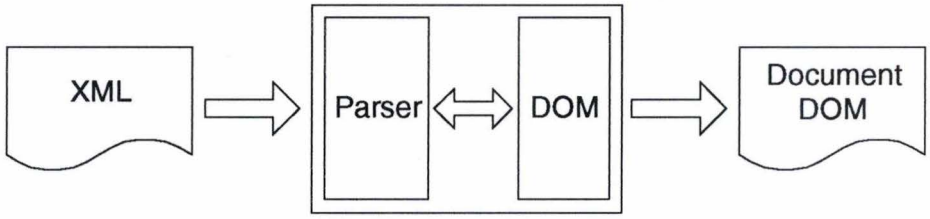

Figure 5.1: la génération du document DOM

Une fois que nous avons l'arbre DOM modélisant l'action, nous pouvons la passer au deuxième module qui va générer l'interface graphique d'édition et de déclenchement de l'action (cfr. figure 5.2). Ce second module est divisé en deux grandes parties : le moteur ICM et le moteur GUI. En effet, pour générer une interface à partir d'une modélisation, on peut distinguer deux fonctions principales : premièrement l'analyse de la modélisation, ensuite la gestion des composants graphiques et des événements de l'interface générée. Le moteur GUI peut-être utilisé indépendamment de notre problématique car il s'agit d'un outil de gestion purement graphique. Le moteur ICM quant à lui, analyse l'arbre DOM et ajoute, grâce au moteur GUI, les composants graphiques ainsi que leurs réactions aux événements à l'interface graphique.

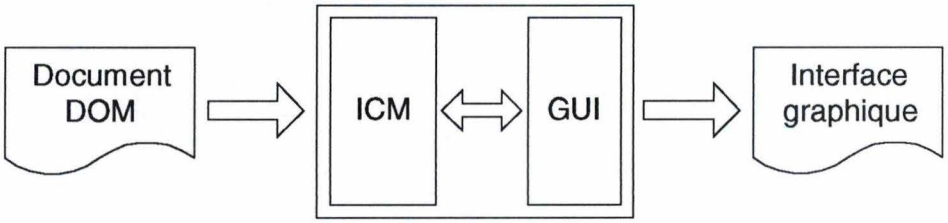

Figure 5.2: la génération de l'interface

A notre sens, cette architecture présente les avantages suivants:

- Le fait d'avoir séparé le programme en deux modules va nous permettre d'avoir une totale indépendance vis-à-vis du *parser*. En effet, on peut facilement changer le *parser* du premier module sans pour autant avoir une quelconque influence sur l'autre module. Bien sûr, le nouveau *parser* doit être compatible SAX.
- En construisant entièrement notre propre moteur, nous avons directement et sans adaptation une application compatible avec l'environnement de travail. Par exemple, notre moteur graphique génère et gère directement les ISMBeans (propres à ISM) alors que les moteurs graphiques préexistants génèrent de simples fenêtres.
- En utilisant une structure DOM, on peut très facilement regénérer le fichier XML initial mais aussi faciliter la sauvegarde des jeux de paramètres. En effet, il suffit d'ajouter à l'arbre DOM modélisant l'action en cours, les noeuds représentant la valeur des paramètres et ensuite d'appeler une méthode qui se charge de retranscrire l'arbre DOM sous forme d'un fichier XML. Il faut aussi remarquer que l'utilisation de DOM ne pose aucun problème au niveau de la disponibilité de la mémoire puisque la modélisation des actions en XML génère des fichiers XML relativement petits.
- Nous disposons d'une totale maîtrise du code et ne dépendons pas de tierce personne pour l'adaptation et l'évolution de certaines fonctions. En un mot, nous disposons des codes sources du moteur et avons tous les droits de modification.
- On ne génère qu'un sous-ensemble restreint d'interfaces graphiques correspondant aux besoins directes de ICM, ce qui permet d'avoir un moteur sans doute moins général qu'un générateur d'interfaces comme TaskGuide Viewer mais bien plus optimisé, simple dans le cadre de notre problématique. Nous tenons tout de même à signaler que le moteur GUI permet de faire bien plus de choses que ce qui est exploité dans le cadre de cette problématique.

5.2.2. Le moteur graphique "GUI"

5.2.2.1. Introduction

La problématique nous impose de générer des interfaces graphiques et de gérer les composants qui les composent. La principale difficulté est le côté dynamique du problème: on veut permettre aux méthodes associées aux composants de pouvoir accéder aux valeurs d'autres composants uniquement sur base de leur identifiant. On veut aussi permettre d'ajouter, supprimer, déplacer des composants en cours

d'exécution. Il nous faut donc un outil nous permettant d'accéder à n'importe quel composant de la fenêtre afin, par exemple, de récupérer la valeur d'un paramètre, lancer une exécution, ... Il s'agit en quelque sorte de mémoriser les objets ajoutés dans la fenêtre et d'y associer de l'information complémentaire (identifiant, procédure associée en réaction à un événement, ...). Nous avons donc créé la classe GUI qui est en quelque sorte le récipient des objets qu'on lui ajoute. Lors de la création de la classe, on choisit le type de récipient: une fenêtre, un *bean*, ou un panneau. Un panneau est un composant qui regroupe un ensemble de composants affichables à l'écran.

Lorsqu'on ajoute une instance de la classe GUI de type fenêtre ou *bean* à une autre instance de la classe GUI de type fenêtre ou *bean*, on chaîne les fenêtres ou *bean* comme dans un *wizard*. Cela permet à partir d'une fenêtre d'accéder aux valeurs de composants d'une autre fenêtre uniquement sur base d'un identifiant logique et sans devoir directement faire appel à une méthode de l'objet concerné. Prenons par exemple, le cas où nous créons une instance de la classe GUI de type fenêtre, nommée "identité" et lui ajoutons des composants dont un champ texte avec comme identifiant "age". Nous créons une autre instance de type fenêtre nommée "permis" à laquelle nous ajoutons une série de composants dont un bouton nommé "continue". Nous ajoutons ensuite la GUI "permis" à la GUI "identité". Nous pouvons maintenant à partir d'une méthode de la fenêtre "permis" accéder à tous les composants de la fenêtre "identité" sur simple identifiant. Nous pouvons donc, lorsque le bouton "continue" est pressé, accéder au champ texte "age" avant de continuer l'exécution.

Une instance de la classe GUI de type panneau peut être ajoutée à une autre instance de la classe GUI. Prenons par exemple, le cas où nous créons une instance de la classe GUI de type panneau nommée "moyenne" et lui ajoutons deux champs textes "revenu1" et "revenu2". Nous assignons comme valeur au panneau la moyenne des champs textes. Lorsqu'un autre GUI demande la valeur du composant "moyenne", il recevra directement la moyenne des valeurs introduites dans les champs textes.

Afin de gérer les composants ajoutés, l'instance GUI gère une liste d'informations. Chaque élément de la liste représente l'information complète concernant un composant ajouté à la fenêtre. L'information est composée de plusieurs champs indiquant : l'identifiant du composant, le type de composant, le label éventuel du composant, un pointeur sur l'élément de la liste représentant son père, une liste de pointeurs sur les éléments de la liste représentant ses fils, éventuellement le nom ou la classe en elle-même ainsi que le nom de la méthode à exécuter en réponse à un événement (*callback*) et un pointeur sur le composant en lui même.

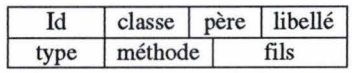

Figure 5.3: informations associées à un composant

Le père d'un composant est souvent le conteneur auquel on a ajouté le composant. Les fils d'un composant (de type conteneur) sont en général les composants qu'on lui ajoute. On attribue à certains composants une méthode d'une classe à appeler lorsqu'un événement se produit. La classe à utiliser peut être donnée, soit sous la forme d'une référence directe à une instance de la classe, soit être le nom de la classe à instancier. Dans le premier cas, on garde un pointeur vers l'objet. Dans le second cas, c'est le moteur qui doit créer l'instance de la classe avant d'utiliser la méthode. Prenons par exemple, le cas où lorsqu'on ajoute un bouton à une instance GUI, on donne l'objet et la méthode associée. Quand le bouton est pressé, la méthode de l'objet sera appelée. Le type du composant indique la nature du composant, par exemple : champ texte, libellé, menu, bar de menu, bouton, boite à cocher, etc.

Cette liste va donc nous permettre un parcours (accès) séquentiel (cfr. figure 5.4) aux composants appartenant directement à ce GUI.

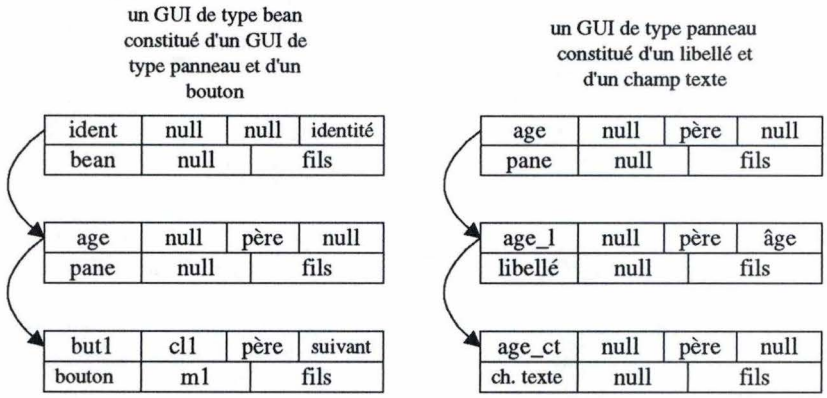

Figure 5.4: la liste des éléments informatifs

Lorsqu'on ajoute une instance de la classe GUI à la classe GUI courante, un seul élément est ajouté à la liste, c'est-à-dire que les deux listes ne sont pas fusionnées. Le GUI ajouté est vu comme un seul composant alors que ce dernier est certainement composé de plusieurs éléments. L'identifiant (local) d'un composant au sein de la liste peut donc être simplement une suite de caractères.

Cette liste permet aussi une recherche sur base du parcours de l'arbre qui représente la hiérarchie des objets (cfr. figure 5.5). Cette hiérarchie est créée par les variables indiquant le père et les fils. Lorsqu'on ajoute une instance de la classe GUI à la classe GUI courante, les deux arborescences sont fusionnées. Nous pouvons accéder à tous les composants appartenant à n'importe quel GUI à partir de n'importe quelle instance de la classe GUI. Naturellement, pour permettre la recherche et l'accès à un composant par parcours de l'arborescence, il faudra utiliser un identifiant un peu plus complexe que la simple suite de caractères de la liste d'éléments. En effet, lorsqu'on ajoute plusieurs instances de la même classe fils GUI à la classe GUI courante, on a dans l'arborescence d'éléments d'informations

plusieurs nœuds ayant le même champ identifiant. L'identifiant d'un composant dans le cas de l'arborescence est composé du champ identifiant de l'élément (comme dans la liste d'éléments) et de celui de tous ses ancêtres. En réalité, l'identifiant de tous ses ancêtres est l'identifiant complet de son père.

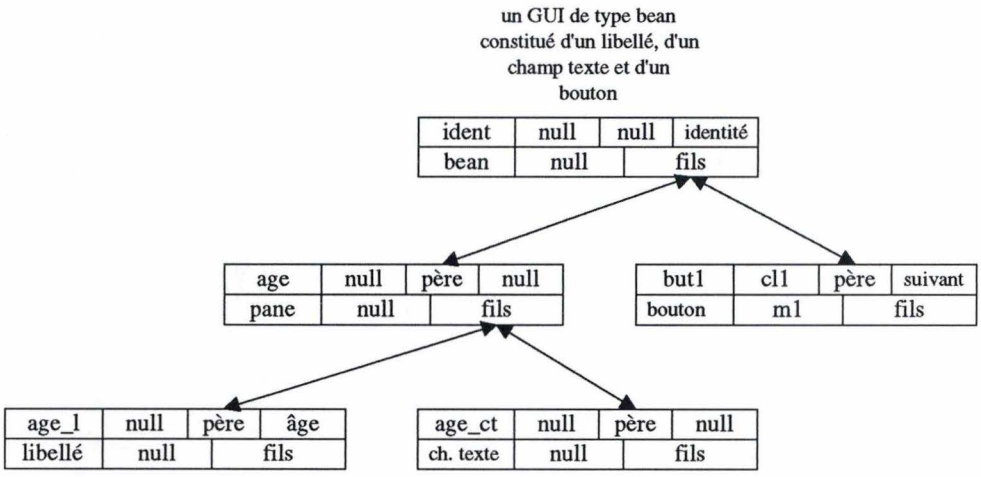

Figure 5.5: l'arbre des éléments informatifs

L'avantage de construire toutes les interfaces à l'aide du moteur est une certaine capacité à évoluer. En effet, si on décide d'utiliser un nouvel ensemble de composants plus performants, ergonomiques, ... il suffit de modifier la classe GUI, et toutes les interfaces construites sur base de cette dernière, après recompilation, bénéficieront des nouveaux composants.

Une manière d'utiliser la classe GUI est de créer un fils de cette classe auquel on a déjà pré-ajouté des composants (via le constructeur). Ensuite, il ne reste plus qu'à définir la méthode addValue() et getValue(). La méthode addValue() permet de gérer la manière dont on veut gérer l'ajout d'une valeur au GUI et la méthode getValue() définit la manière dont on récupère la valeur logique d'un GUI. Prenons par exemple, un composant GUI dont la fonction est de calculer la moyenne des chiffres qu'on lui donne. A la création, le composant GUI est composé d'un titre et d'un libellé. Lorsqu'on ajoute un chiffre, on appelle la méthode addValue(X). Cette dernière, que nous avons définie, ajoute un libellé au composant GUI avec comme intitulé la valeur ajoutée. On répète cette opération un certain nombre de fois, jusqu'au moment où l'on appelle getValue(). Cette dernière calcule et retourne la moyenne des valeurs contenues dans les libellés. Un composant GUI permet donc de pré-grouper un ensemble de composants et de définir la manière dont on gère l'ajout de valeurs, de composants mais aussi la manière de calculer le résultat logique du composant GUI.

5.2.2.2. Les méthodes d'ajout de composants

Nous allons maintenant passer en revue les principales méthodes de la classe GUI. Un premier groupe de méthodes permet d'ajouter les composants au sein de l'instance GUI:

- public void addButton(String myLabel, String ident, Object myCallBackClass, String myMethod, String father)

  Cette méthode permet d'ajouter un bouton à l'instance de la classe GUI avec comme libellé le contenu du paramètre "myLabel", comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Le contenu du paramètre "myCallBackClass" peut, soit être l'instance de l'objet, soit être le nom de la classe qui possède une méthode à utiliser lorsque le bouton est pressé. Cette méthode est identifiée par le contenu du paramètre "myMethod".

- public void addCheckBox(String labelText, String ident, String father)

  Cette méthode permet d'ajouter une boite à cocher à l'instance de la classe GUI avec comme libellé le contenu du paramètre "labelText", comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père.

- public void addComponent(Component myCompo, String ident, String father)

  Cette méthode permet d'ajouter le composant contenu dans le paramètre "myCompo" à l'instance de la classe GUI avec comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Aucun événement généré par ce composant ne sera géré par la classe. Cette méthode permet d'insérer des éléments de type purement graphique telle qu'une barre de séparation, ...

- public void addHelpMenu(String myLabel, String ident, String father, Object myCallBackClass, String myMethod)

  Cette méthode permet d'ajouter, dans la barre des menus, un menu d'aide à l'instance de la classe GUI avec comme libellé le contenu du paramètre "myLabel", comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Le contenu du paramètre "myCallBackClass" peut, soit être l'instance de l'objet, soit être le nom de la classe qui possède une méthode à utiliser lorsque le menu d'aide est pressé. Cette méthode est identifiée par le contenu du paramètre "myMethod".

- public void addLabel(String labelText, String ident, String father)

  Cette méthode permet d'ajouter un libellé à l'instance de la classe GUI avec comme texte le contenu du paramètre "labelText", comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père.

- public void addMenu(String myLabel, String ident, String father, Object myCallBackClass, String myMethod)

  Cette méthode permet d'ajouter, dans la barre des menus, un menu à l'instance de la classe GUI avec comme libellé le contenu du paramètre "myLabel", comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Le contenu du paramètre "myCallBackClass" peut, soit être l'instance de l'objet, soit être le nom de la classe qui possède une méthode à utiliser lorsque le menu est pressé. Cette méthode est identifiée par le contenu du paramètre "myMethod".

- public void addMenuItem(String myLabel, String ident, String father, Object myCallBackClass, String myMethod)

  Cette méthode permet d'ajouter un *item* à un menu de l'instance de la classe GUI avec comme libellé le contenu du paramètre "myLabel", comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Le contenu du paramètre "myCallBackClass" peut, soit être l'instance de l'objet, soit être le nom de la classe qui possède une méthode à utiliser lorsque l'*item* du menu est pressé. Cette méthode est identifiée par le contenu du paramètre "myMethod".

- public void addMultiChoiceList(String listOfItem[],String ident, String father)

  Cette méthode permet d'ajouter une liste multi-choix à l'instance de la classe GUI avec comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Le paramètre "listOfItem" contient la liste des choix possibles.

- public void addPane(GUI GUIPane, String father)

  Cette méthode permet, soit de chaîner deux instances de GUI de type fenêtre ou *bean*, soit d'ajouter une instance de GUI de type panneau à une autre instance de GUI. Le contenu du paramètre "father" identifie le composant père auquel on ajoute l'instance GUI. Les arborescences des éléments des deux GUIs sont fusionnées.

- public void addPanel(String ident, String father)

  Cette méthode permet d'ajouter un panneau à l'instance de la classe GUI avec comme identifiant le contenu du paramètre "ident" et le contenu du paramètre "father" comme identifiant du père. Un panneau est un composant qui permet de regrouper des composants.

- public void addPassword(String ident, String father)

  Cette méthode permet d'ajouter un champ texte de type mot de passe (les caractères encodés ne sont pas affichés) à l'instance de la classe GUI avec comme identifiant le contenu du paramètre "ident" et le contenu du paramètre "father" comme identifiant du père.

- public void addSimpleChoiceList(String lstItem[],String ident, String father)

  Cette méthode permet d'ajouter une liste à choix unique à l'instance de la classe GUI avec comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Le paramètre "lstItem" contient la liste des choix possibles.

- public void addTab(String myLabels[],String myPages[],String ident, String father)

  Cette méthode permet d'ajouter des onglets à l'instance de la classe GUI avec comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Le paramètre "myLabels" est un tableau contenant les libellés des onglets. Le paramètre "myPages" est un tableau contenant les identifiants de chaque page associée à un onglet. Ce sont ces identifiants qui seront utilisés pour ajouter les composants associés à un onglet.

- public void addTextArea(String content, String ident, String father)

  Cette méthode permet d'ajouter une zone texte libre à l'instance de la classe GUI avec comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père. Une zone libre permet l'encodage de textes longs. La valeur du paramètre "content" est le contenu par défaut de la zone texte.

- public void addTextField(String ident, String father)

  Cette méthode permet d'ajouter un champ texte à l'instance de la classe GUI avec comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du père.

- public void addValue(Object Value)

  Cette méthode est à définir par l'utilisateur lors de la création d'un fils spécialisé de la classe GUI. Cette méthode définit la manière de gérer l'ajout d'une valeur logique ou d'un composant.

- public void addValueChoice(Object value, String ident)

  Cette méthode permet d'ajouter la valeur du paramètre "value" à la liste de choix identifiée par le contenu du paramètre "ident".

Naturellement, il existe plusieurs variantes des méthodes présentées précédemment, notamment pour spécifier la position relative des composants, pour spécifier une icône associée, etc. :

- public void addButton(String myLabel, Icon icon, int x, int y, String ident, Object myCallBackClass, String myMethod, String father)

  Cette méthode permet d'ajouter un bouton à l'instance de la classe GUI avec comme libellé le contenu du paramètre "myLabel", comme identifiant le contenu du paramètre "ident", et le contenu du paramètre "father" comme identifiant du

père. Le contenu du paramètre "myCallBackClass" peut, soit être l'instance de l'objet, soit être le nom de la classe qui possède une méthode à utiliser lorsque le bouton est pressé. Cette méthode est identifiée par le contenu du paramètre "myMethod". Le paramètre "icon" contient l'icône du bouton. Le contenu du paramètre "x" et "y" forment la coordonnée du composant dans le *layout* (gestionnaire Java de mise en page) du conteneur : l'espace affichable est considéré comme une grille où chaque coordonnée représente une cellule de la grille.

5.2.2.3. Les méthodes de manipulation des composants

Nous allons maintenant passer en revue les méthodes qui permettent d'accéder aux objets du conteneur GUI et de les manipuler:

- public GUI findGUIChild(String GUIIdent)

  Cette méthode permet de chercher et d'obtenir, parmi les composants descendants de l'instance courante du GUI, le composant GUI identifié par le contenu du paramètre "GUIIdent".

- public GUI findGUIFather(String GUIIdent)

  Cette méthode permet de chercher et d'obtenir, parmi les composants ancêtres de l'instance courante du GUI, le composant GUI identifié par le contenu du paramètre "GUIIdent".

- public windowElements getElement(String Ident)

  Cette méthode permet de chercher et d'obtenir dans la liste interne de gestion des composants, l'élément identifié par le contenu du paramètre "Ident".

- public GUI getGUI(String GUIIdent)

  Cette méthode permet de chercher et d'obtenir le composant GUI identifié par le contenu du paramètre "GUIIdent". Cette méthode de recherche parcourt toute l'arborescence.

- public Object getMenu(String ident)

  Cette méthode permet de chercher et d'obtenir dans la liste interne de gestion des composants, l'objet de type menu identifié par le contenu du paramètre "ident".

- public Object getMenuBar()

  Cette méthode permet de chercher et d'obtenir dans la liste interne de gestion des composants, la barre de menu.

- public Object getObject(String ident)

  Cette méthode permet de chercher et d'obtenir dans la liste interne de gestion des composants, le composant identifié par le contenu du paramètre "ident".

- public Object getObject(String ident, String GUIIdent)

  Cette méthode permet de chercher et d'obtenir dans la liste interne de gestion des composants du GUI identifié par le contenu du paramètre "GUIIdent", le composant identifié par le contenu du paramètre "ident".

- public Object getValue()

  Cette méthode définit la manière de calculer la valeur résultat d'un GUI. Cette méthode est très utile pour les GUIs de type panneau lorsqu'il faut définir leur valeur retour.

- public Object getValue(String Ident)

  Cette méthode renvoie la valeur courante du composant identifié par le contenu du paramètre "Ident". S'il s'agit d'un composant de type GUI, alors il fait appel à la méthode getValue() de ce dernier pour calculer la valeur résultat, sinon il accède directement à la valeur courante du composant.

- public void remove(String ident)

  Cette méthode supprime du GUI le composant identifié par le contenu du paramètre "ident".

- public void remove(windowElements myElement)

  Cette méthode supprime du GUI le composant associé à l'élément, contenu dans le paramètre "myElement", de la liste de gestion des composants ainsi que tous ses fils.

- public void removeChild(windowElements myElement)

  Cette méthode supprime du GUI tous les composants fils du composant associé à l'élément, contenu dans le paramètre "myElement", de la liste de gestion des composants.

- public void removeAll()

  Cette méthode supprime tous les composants du GUI ainsi que tous les GUIs chaînés en amont ou en aval du GUI courant.

- public void setLayout(LayoutManager myLayout, String ident)

  Cette méthode permet d'assigner le gestionnaire de composants, contenu dans le paramètre "myLayout", au conteneur identifié par le contenu du paramètre "ident".

- public void setTitle(String myTitle)

  Cette méthode permet de définir le titre du GUI comme étant le contenu du paramètre "myTitle". Cette méthode est à utiliser dans les GUIs de type fenêtre ou *bean*.

- public void hide()

  Cette méthode permet de cacher (ne plus afficher) le GUI, si ce dernier est de type fenêtre ou *bean.*

- public void show()

  Cette méthode permet d'afficher le GUI, si ce dernier est de type fenêtre ou *bean.*

- public void hide(String myGUI)

  Cette méthode permet de cacher (ne plus afficher) le GUI identifié par le contenu du paramètre "myGUI" à condition que ce dernier est de type fenêtre ou *bean.* Cette méthode est très utile dans la gestion de l'enchaînement des fenêtres, notamment pour la conception de *wizards.*

- public void show(String myGUI)

  Cette méthode permet d'afficher le GUI identifié par le contenu du paramètre "myGUI", à condition que ce dernier est de type fenêtre ou *bean.* Cette méthode est très utile dans la gestion de l'enchaînement des fenêtres notamment pour la conception de *wizards.*

5.2.3. Le moteur "ICM"

Le moteur ICM est relativement simple puisqu'il ne fait qu'assembler au fur et à mesure un ensemble de fils de la classe GUI. Nous tenons à faire la remarque préliminaire suivante : chaque fois que nous parlons d'ajouter une valeur, cette dernière est ajoutée dans la langue du navigateur grâce à l'appel à une méthode qui gère les fichiers *properties.*

L'instance principale de la classe GUI est de type *bean.* C'est elle qui sera insérée et affichée dans la partie droite du Webtop. Elle est donc le noeud père de tous les composants et de tous les GUIs. Nous commençons par lui assigner comme titre le libellé de l'action et ensuite nous lui ajoutons un menu de jeux de paramètres composé des éléments "sauver" et "sauvegarder". A l'élément "sauver" est associé une méthode qui modifie l'arbre DOM de l'action courante pour lui ajouter les valeurs du jeu de paramètres (cfr. modélisation). Ensuite l'arbre DOM génère un fichier XML qui est sauvegardé sur le *framework* grâce à l'*archiver*. Pour recharger un jeu de paramètres, il suffit de récupérer grâce à l'*archiver* le fichier XML qui modélise l'action, le jeu de paramètres et ensuite de réappeler l'application comme s'il s'agissait d'une simple action.

S'il existe une aide à l'action, nous créons une instance de la classe ICMHelp (classe fils de la classe GUI composée d'une zone texte et d'un bouton de clôture de l'affichage de la fenêtre) et l'associons au menu d'aide.

Maintenant, il ne reste plus qu'à ajouter tous les paramètres les uns en-dessous des autres. Nous avons créé une classe fils de la classe GUI pour chaque type de paramètres : champ texte, simple choix, multi-choix. Naturellement, chacun des paramètres peut posséder une valeur par défaut. Nous allons passer rapidement en revue les différentes classes sans trop nous attarder sur toutes les fonctionnalités offertes par celles-ci:

- la classe "ICMTextField" est une classe fille de la classe GUI qui représente un paramètre de type champ texte. Elle est composée d'un libellé et d'un champ texte. Lorsqu'on utilise la méthode getValue(), elle renvoie la valeur du champ texte.
- la classe "ICMSimpleChoice" est une classe fille de la classe GUI qui représente un paramètre de type simple choix. Elle est composée d'un libellé et d'un bouton. Lorsqu'on appuie sur le bouton, une fenêtre apparaît, affichant tous les choix possibles sous forme de boutons radios ainsi qu'un bouton de fermeture. Le constructeur utilisé pour créer l'instance de la classe indique si les valeurs sont données ou s'il faut les calculer à l'aide d'une fonction. La méthode addValue(X) est utilisée pour ajouter les valeurs données ou calculées à la liste de choix. Lorsqu'on utilise la méthode getValue(), elle renvoie l'élément sélectionné.
- la classe "ICMMultiChoice" est une classe fille de la classe GUI qui représente un paramètre de type multi-choix. Elle est composée d'un libellé et d'un bouton. Lorsqu'on appuie sur le bouton, une fenêtre apparaît, affichant tous les choix possibles sous forme de boîtes à cocher ainsi qu'un bouton de fermeture. Le constructeur utilisé pour créer l'instance de la classe, indique si les valeurs sont données ou s'il faut les calculer à l'aide d'une fonction. La méthode addValue(X) est utilisée pour ajouter les valeurs données ou calculées à la liste de choix . Lorsqu'on utilise la méthode getValue(), elle renvoie le ou les élément(s) sélectionné(s).

Lorsque tous les paramètres sont ajoutés, il ne reste plus qu'à ajouter une instance de la classe "ICMSimpleChoice" contenant tous les modes d'exécution disponibles ainsi qu'une instance de la classe "ICMOptionExecution". Cette classe est une classe fille de la classe GUI composée de deux boutons : un pour annuler l'exécution et l'autre pour la lancer. Avant de lancer l'exécution, la classe vérifie que tous les paramètres obligatoires ont une valeur et génère sinon une fenêtre d'avertissement.

L'exécution des fonctions de calcul de valeurs possibles d'un paramètre et d'exécution de l'action nécessite l'utilisation du SML Wrapper. Certaines valeurs des paramètres de la fonction SML peuvent être la valeur d'autres paramètres de l'action. C'est à ce niveau que les fonctionnalités offertes par la classe GUI (arborescence et chaînage) sont très utiles pour récupérer la valeur d'un paramètre sur base de son identifiant.

6. Conclusion

Comme nous le pensions, le prototype réalisé nous a permis de bien jauger la difficulté et les avantages des solutions de type maison.

Tout d'abord, ce prototype nous a permis de nous initier et finalement maîtriser les technologies liées à Java et XML. Ces connaissances seront très utiles lors de l'évaluation et du choix de la solution la plus adéquate pour ICM. En effet, il est maintenant plus facile d'estimer la difficulté de développer une solution complète. Le développement de ce prototype a aussi permis d'apporter à l'équipe d'ICM un savoir faire réutilisable pour de futures développements. Le prototype réalisé possède les grandes qualités suivantes:

- il s'adapte parfaitement à l'environnement Webtop;
- il a des performances relativement meilleures que les autres moteurs. Nous tenons surtout à souligner sa vitesse d'exécution et de génération de l'interface. En effet, les autres moteurs (non encore adaptés à l'environnement de fonctionnement) sont plus lents;
- il tient dans un fichier compressé au format jar de106k.

Comme nous avons pu le constater, XML est un langage relativement simple tant au niveau de sa conception que de son utilisation. XML est un métalangage de définitions de formats de documents initialement conçu pour l'échange de données. Néanmoins, il convient parfaitement pour la modélisation de nos actions et à fortiori à d'autres modélisations. De plus, les normes DOM et SAX uniformisent la manière d'utiliser XML. Elles permettent de rendre les applications tels que les *parsers* interchangeables et elles aident les programmeurs à plus facilement appréhender les programmes des autres.

Dans ce mémoire, nous vous avons présenté la problématique, les solutions envisageables et enfin la solution implémentée. La solution implémentée, à savoir, la modélisation de l'action et le moteur de génération d'interfaces, n'est qu'un prototype qui par définition, devra être complété et enrichi de certaines fonctionnalités.

En premier lieu, il faudrait adapter la modélisation de l'action et le moteur afin de permettre la gestion d'une séquence d'actions. Toutefois, lors de la création de la classe de gestion de composants (la classe GUI), nous l'avons construite de façon à pouvoir facilement chaîner des fenêtres (faire apparaître successivement les fenêtres comme dans un *wizard* d'installation d'une application) et permettre un accès facile à

leurs composants. Une autre manière envisagée pour la gestion des séquences d'actions était de placer chaque action dans un onglet. A nouveau, la classe GUI permet la gestion de ces derniers. En ce qui concerne la modélisation de la séquence, nous avons laissé une porte ouverte dans la DTD et, le moteur ICM basé sur l'analyse de l'arbre DOM est facilement adaptable.

Ensuite, il faut encore développer et implémenter une grande partie des fonctionnalités sur l'arborescence des composants. En effet, lors de l'élaboration de la classe GUI, nous avons porté principalement l'accent sur les méthodes basées sur le parcours de la liste d'éléments. Ce sont de ces dernières que nous avions directement besoin pour la génération de l'interface de l'action. Nous avons commencé à implémenter quelques méthodes sur l'arborescence, notamment l'affichage d'un GUI spécifié, mais il faut encore développer la gestion des identifiants, la recherche d'un composant dans l'arborescence (plus dans la liste d'éléments locaux), etc. De toute façon, comme la manière principale d'utiliser la classe est de créer un fils, il suffit de recompiler les fils après avoir modifié le père, et donc à priori, il n'y aura absolument rien à modifier dans les interfaces déjà développées à l'aide de la classe GUI.

Dans la modélisation XML, nous n'avons envisagé qu'un seul fichier *properties* par langue, pour l'ensemble des actions. On peut facilement envisager d'ajouter à la modélisation d'une action un *tag* ou un attribut définissant les fichiers *properties* propres à l'action. Cela permettrait une plus grande indépendance des actions les unes par rapport aux autres.

Ensuite, il faut encore développer une certaine interactivité entre l'arbre AGOV (arbre du Webtop) et notre application. En effet, la gestion des valeurs des paramètres pourrait être directement fournie par l'arbre AGOV. Par exemple, le domaine d'une action pourrait être déterminé dans l'arbre AGOV et récupéré par notre application lors du déclenchement de l'action. L'arbre pourrait être non seulement utilisé lors de l'édition de l'action mais aussi lors de l'ajout de nouvelles actions. En effet, les versions ultérieures de l'arbre AGOV laissent espérer pouvoir éditer directement de manière graphique les fichiers XML en son sein. Il ne faudrait donc plus les éditer, les ajouter manuellement dans l'arbre AGOV et sur le *framework*.

Enfin, comme nous l'avons déjà dit, nous avons la modélisation de l'action et la capacité de la déclencher. Maintenant, nous avons besoin d'une modélisation du résultat de l'action. En effet, le déroulement d'une action peut rencontrer quelques problèmes. Prenons par exemple, le cas où l'on agrandit de 20 MB, le *swap* sur un disque. Il y aura un problème s'il n'y a pas assez de place sur le disque. Comme il existe un grand nombre d'agents, de services sur le *framework*, la gestion des comptes rendus d'exécution est très difficile puisque chacun les exprime à sa manière. Donc, afin de faciliter la gestion et l'affichage des comptes rendus d'exécution, il faudrait définir une modélisation XML commune à tous.

leurs composants. Une autre manière envisagée pour la gestion des séquences d'actions était de placer chaque action dans un onglet. A nouveau, la classe GUI permet la gestion de ces derniers. En ce qui concerne la modélisation de la séquence, nous avons laissé une porte ouverte dans la DTD et, le moteur ICM basé sur l'analyse de l'arbre DOM est facilement adaptable.

Ensuite, il faut encore développer et implémenter une grande partie des fonctionnalités sur l'arborescence des composants. En effet, lors de l'élaboration de la classe GUI, nous avons porté principalement l'accent sur les méthodes basées sur le parcours de la liste d'éléments. Ce sont de ces dernières que nous avions directement besoin pour la génération de l'interface de l'action. Nous avons commencé à implémenter quelques méthodes sur l'arborescence, notamment l'affichage d'un GUI spécifié, mais il faut encore développer la gestion des identifiants, la recherche d'un composant dans l'arborescence (plus dans la liste d'éléments locaux), etc. De toute façon, comme la manière principale d'utiliser la classe est de créer un fils, il suffit de recompiler les fils après avoir modifié le père, et donc à priori, il n'y aura absolument rien à modifier dans les interfaces déjà développées à l'aide de la classe GUI.

Dans la modélisation XML, nous n'avons envisagé qu'un seul fichier *properties* par langue, pour l'ensemble des actions. On peut facilement envisager d'ajouter à la modélisation d'une action un *tag* ou un attribut définissant les fichiers *properties* propres à l'action. Cela permettrait une plus grande indépendance des actions les unes par rapport aux autres.

Ensuite, il faut encore développer une certaine interactivité entre l'arbre AGOV (arbre du Webtop) et notre application. En effet, la gestion des valeurs des paramètres pourrait être directement fournie par l'arbre AGOV. Par exemple, le domaine d'une action pourrait être déterminé dans l'arbre AGOV et récupéré par notre application lors du déclenchement de l'action. L'arbre pourrait être non seulement utilisé lors de l'édition de l'action mais aussi lors de l'ajout de nouvelles actions. En effet, les versions ultérieures de l'arbre AGOV laissent espérer pouvoir éditer directement de manière graphique les fichiers XML en son sein. Il ne faudrait donc plus les éditer, les ajouter manuellement dans l'arbre AGOV et sur le *framework*.

Enfin, comme nous l'avons déjà dit, nous avons la modélisation de l'action et la capacité de la déclencher. Maintenant, nous avons besoin d'une modélisation du résultat de l'action. En effet, le déroulement d'une action peut rencontrer quelques problèmes. Prenons par exemple, le cas où l'on agrandit de 20 MB, le *swap* sur un disque. Il y aura un problème s'il n'y a pas assez de place sur le disque. Comme il existe un grand nombre d'agents, de services sur le *framework*, la gestion des comptes rendus d'exécution est très difficile puisque chacun les exprime à sa manière. Donc, afin de faciliter la gestion et l'affichage des comptes rendus d'exécution, il faudrait définir une modélisation XML commune à tous.

7. Bibliographie

Références Web

Aelfred	Http://www.microstar.com/XML
Document Object Model	Http://www.w3.org/DOM/
Extensible Stylesheet Language	Http://www.w3.org/TR/WD-xsl
JavaBean	Http://java.sun.com/beans
Prototype	Http://www.cam.org/~pierlou/prototype
SAX	Http://www.megginson.com/SAX
TaskGuide viewer	Http://www.alphaworks.ibm.com/formula/taskguideviewer

Livre

Titre	Auteur	Maison d'édition
Java	Laura Lemay & Charles L. Perlins	Simon & Schuster Maemillan 1996

Documents propres à Bull

Titre	Auteur
Spécifications du langage d'administration AXAML	Cécile Baille-Pierre
ISM Configuration Manager	Christian Ritter

8. Abréviations

DOM	Document Object Model
DTD	Document Type Definition
ICM	ISM Configuration Manager
ISM	Integrated System Management
SML	System Management Language
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language

9. Annexe

9.1. DTD ICM ACTION

```
<!ENTITY % boolean "(true|false)">

<!ENTITY % typeParameter
"TEXTFIELD|SIMPLECHOICE|MULTICHOICE|TEXTAREA">

<!ELEMENT icm (action|listAction)>

<!ELEMENT action
(listParameterSetDescription?,help,parameterList,optionExecution)>
<!ATTLIST action
 idAction          ID       #REQUIRED
 type              CDATA    #REQUIRED
 propertiesLabel   ID       #IMPLIED
 label             CDATA    #IMPLIED
>

<!ELEMENT listParameterSetDescription (#PCDATA)>

<!ELEMENT help (#PCDATA)>
<!ATTLIST help
 isProperties      (%boolean;)     #REQUIRED
>

<!ELEMENT parameterList (parameter)*>

<!ELEMENT parameter (defaultValue?,(listValue|function)?,parameterSet?)>
<!ATTLIST parameter
 idParameter       ID                  #REQUIRED
 propertiesLabel   ID                  #IMPLIED
 label             CDATA               #IMPLIED
 type              (%typeParameter;)   #REQUIRED
 mandatory         (%boolean;)         #REQUIRED>

<!ELEMENT listValue (value)+>

<!ELEMENT defaultValue (value)+>

<!ELEMENT parameterSet (value)+>
```

```
<!ELEMENT function    (parameterFunction)*>
<!ATTLIST function
 name                 ID                  #REQUIRED
>

<!ELEMENT parameterFunction (value)+>
<!ATTLIST parameterFunction
 isIdParameter        (%boolean;)        #REQUIRED >

<!ELEMENT value       (#PCDATA)>
<!ATTLIST value
 isProperties         (%boolean;)        #REQUIRED
 returnValue          CDATA              #IMPLIED
>

<!ELEMENT optionExecution
(constraintCheck?,execute?,execCheck?,rollBack?)>

<!ELEMENT constraintCheck   (function)>

<!ELEMENT execute           (function)>

<!ELEMENT execCheck         (function)>

<!ELEMENT rollBack          (function)>

<!ELEMENT listAction        (#PCDATA)>
```

9.2. TAGS DE TASKGUIDE VIEWER

Tag Name	Attributes
<a>	action, cancel, goto, href, quit, target
<al>	left, right
<b>	No Attributes
<back-button>	action, goto, quit
 	No Attributes
<cancel-button>	action, goto, quit
<cite>	No Attributes
<dataentry>	convert-spaces, format, init, link-to, multiple, name, no-indent, password, reinit, required, size
<editlist>	add, add-button, delete, delete-button, edit, edit-button, first, format, headers, init, invisible,key, last, minimum-rows, name, no-add, no-delete, no-edit, ordered, reinit, reselect, select, select-policy, separator, sort, splitbar, type
<extra-button>	action, goto
<file-field>	file-spec, init, name, open, reinit, save-as
<help-button>	action, goto
<i>	No Attributes
<img>	No Attributes
	action, goto, quit
<next-button>	action, goto, quit
	alpha, alphaLower, roman, romanLower, style
<option>	fill-in, icon-id, link-name, link-to-name, selected, set-image, set-next, unselected-value, value
<p>	No Attributes
<panel>	add-panel-n, back, cancel-url, cancel-frame, disable-back, disable-cancel, disable-extra, disable-help, disable-next, disable-undo, enable-back, enable-cancel, enable-extra, enable-help, enable-next, enable-undo, error-panel, help-frame, help-panel-id, help-lib, help-style, help-url, hide-back, hide-cancel, hide-extra, hide-help, hide-next, hide-undo, image-file, in-toc, name, next, no-image, not-in-notebook, not-in-toc, remove-panel-n, text-font, text-size, title-font, title-size
<pre>	No Attributes
<plug-in>	dll, name
<select>	combo-box, drop-down, link-to, list-box, name, required, several, simple-combo
<sguide>	Cancel-url, cancel-frame, disable-back, Disable-cancel, disable-extra, disable-help, disable-next, disable-undo, dont-save-values-on-exit, enable-back, enable-cancel, enable-extra, enable-help, enable-next, enable-undo, first panel, help-frame, help-id, help-lib, help-style, help-url, hide-back, hide-cancel, hide-extra, hide-help, hide-next, hide-undo, high-res-size, image-file, image-id, image-lib, init, low-res-size, med-res-size, notebook, save-values-on-exit,

	show-back, show-cancel, show-extra, show-help, show-next, show-undo, text-font, text-size, title-font, title-size, toc, use-saved-values
<sl>	No Attributes
<table>	colspec, no-border, no-horizontal-lines, no-lines, no-vertical-lines
<td>	No Attributes
<th>	No Attributes
<title>	No Attributes
<tr>	No Attributes
<tt>	No Attributes
<u>	No Attributes
	No Attributes
<undo-buton>	action, goto

9.3. DTD PROTOTYPE

```
<!ENTITY % languages "(js | xml | html | tcl | bean | proto )">
<!ENTITY % truefalse "(true | false | yes | no | on | off | 1 | 0)">
<!ENTITY % events " onAction? | onAdjustment? | onComponent* | onContainer* |
onFocus* | onItem? | onKey* | onMouse* | onMouseMotion* | onText? ">
<!ENTITY % sborders " lineBorder | doubleLineBorder | etchedBorder |
bevelBorder | emptyBorder ">
<!ENTITY % borders "%sborders; | titledBorder | compoundBorder ">
<!ENTITY % fontstyle "(plain | bold | italic | boldItalic | italicBold)">
<!ENTITY % colors "(black | blue | cyan | darkGray | gray | green | lightGray |
magenta | orange | pink | red | white | yellow)">
<!ENTITY % panes " directorypane? | htmlpane? | tabpane? ">
<!ENTITY % buttons " radiobutton* | checkbox* | togglebutton* ">
<!ENTITY % choosers " colorchooser? | filechooser? ">
<!ENTITY % widgets " label* | text* | password* | textarea* | listbox* | combobox*
| grid* | slider* | scrollbar* | progressbar* | tree* ">
<!ENTITY % controls " container* | (%widgets;)* | (%choosers;)* | (%buttons;)* |
button* | groupbutton* | (%panes;)* ">
<!ENTITY % dialogboxes " messageBox | confirmBox | optionBox | inputBox ">
<!ENTITY % dialogTypes "(error | information | question | warning | plain)">
<!ENTITY % dialogOptions "(yes_no | yes_no_cancel | ok_cancel | default)">

<!ELEMENT Prototype ( (window | panel )? )>
<!ATTLIST Prototype
      ID CDATA #IMPLIED>

<!ELEMENT Window ( (onCode* | onWindow*)* , ( panel* |
            (%dialogboxes;)* | database? )* )>
<!ATTLIST Window
      ID CDATA #IMPLIED
      TOP CDATA #IMPLIED
      LEFT CDATA #IMPLIED
      HEIGHT CDATA #IMPLIED
      WIDTH CDATA #IMPLIED
      TITLE CDATA #IMPLIED
      ICON CDATA #IMPLIED>
<!ATTLIST Window UI (WINDOWS | MOTIF | JAVA |
            SANTAFE | VANCOUVER) "WINDOWS">
<!ATTLIST Window VISIBLE %truefalse; "TRUE">
<!ATTLIST Window RESIZABLE %truefalse; "TRUE">

<!ELEMENT panel (( onCode* | (%events;)* | (%borders;)? | menu? |
```

```
            toolbar? )*, ( frameset | (%controls;)* )?)>
<!ATTLIST panel
        ID CDATA #IMPLIED
        TABLEREF CDATA #IMPLIED
        TIP CDATA #IMPLIED>
<!ATTLIST panel BACKGROUND %colors; "LIGHTGRAY">

<!ELEMENT menu ((%borders;)?|menubar+)*>
<!ATTLIST menu
        ID CDATA #IMPLIED>

<!ELEMENT menubar (( menuitem | menucheckbox | menuradiobutton |
            menubar | menugroupbutton) | (%events;)* |
            font? | (%borders;)?)+>
<!ATTLIST menubar
        ID CDATA #IMPLIED
        CAPTION CDATA #REQUIRED
        SHORTCUT CDATA #IMPLIED>
<!ATTLIST menubar ENABLED %truefalse; "TRUE">
<!ATTLIST menubar BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST menubar FOREGROUND %colors; "BLACK">

<!ELEMENT menugroupbutton (menucheckbox|menuradiobutton)+>
<!ATTLIST menugroupbutton
        ID CDATA #IMPLIED>

<!ELEMENT menuitem ((%events;)*|font?|(%borders;)?)*>
<!ATTLIST menuitem
        ID CDATA #IMPLIED
        CAPTION CDATA #REQUIRED
        SHORTCUT CDATA #IMPLIED
        ICON CDATA #IMPLIED>
<!ATTLIST menuitem ENABLED %truefalse; "TRUE">
<!ATTLIST menuitem BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST menuitem FOREGROUND %colors; "BLACK">

<!ELEMENT menucheckbox ((%events;)*|font?|(%borders;)?)*>
<!ATTLIST menucheckbox
        ID CDATA #IMPLIED
        CAPTION CDATA #REQUIRED
        SHORTCUT CDATA #IMPLIED>
<!ATTLIST menucheckbox ENABLED %truefalse; "TRUE">
<!ATTLIST menucheckbox SELECTED %truefalse; "FALSE">
<!ATTLIST menucheckbox BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST menucheckbox FOREGROUND %colors; "BLACK">
```

```
<!ELEMENT menuradiobutton ((%events;)*|font?|(%borders;)?)*>
<!ATTLIST menuradiobutton
        ID CDATA #IMPLIED
        CAPTION CDATA #REQUIRED
        SHORTCUT CDATA #IMPLIED>
<!ATTLIST menuradiobutton ENABLED %truefalse; "TRUE">
<!ATTLIST menuradiobutton SELECTED %truefalse; "FALSE">
<!ATTLIST menuradiobutton BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST menuradiobutton FOREGROUND %colors; "BLACK">

<!ELEMENT toolbar ((%events;)* | (%borders;)? | (toolgroupbutton* | tool)*)*>
<!ATTLIST toolbar
        ID CDATA #IMPLIED>
<!ATTLIST toolbar BACKGROUND %colors; "LIGHTGRAY">

<!ELEMENT toolgroupbutton (tool)+>
<!ATTLIST toolgroupbutton
        ID CDATA #IMPLIED>

<!ELEMENT tool ((%events;)* | (%borders;)?)*>
<!ATTLIST tool
        ID CDATA #IMPLIED
        IMAGE CDATA #REQUIRED
        TIP CDATA #IMPLIED>
<!ATTLIST tool TOGGLE %truefalse; "TRUE">
<!ATTLIST tool BACKGROUND %colors; "LIGHTGRAY">

<!ELEMENT frameset ((%events;)* | (%borders;)? | (panel,panel))*>
<!ATTLIST frameset
        ID CDATA #IMPLIED
        POS CDATA #IMPLIED
        PERCENT CDATA #IMPLIED
        DIVIDERSIZE CDATA #IMPLIED>
<!ATTLIST frameset ORIENTATION (HORIZONTAL | VERTICAL )
"VERTICAL">

<!ELEMENT container ( (%events;)* | (%borders;)? | (%controls;)*)*>
<!ATTLIST container
        ID CDATA #IMPLIED
        POS CDATA #IMPLIED
        LOCATION CDATA #IMPLIED
        TIP CDATA #IMPLIED>
<!ATTLIST container BACKGROUND %colors; "LIGHTGRAY">

<!ELEMENT label ( (%events;)* | font? | (%borders;)? )*>
<!ATTLIST label
```

```
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        TIP CDATA #IMPLIED
        IMAGE CDATA #IMPLIED
        CAPTION CDATA #IMPLIED>
<!ATTLIST label JUSTIFICATION (LEFT | CENTER | RIGHT) "LEFT">
<!ATTLIST label BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST label FOREGROUND %colors; "BLACK">

<!ELEMENT text ( (%events;)* | font? | (%borders;)?)*>
<!ATTLIST text
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        VALUE CDATA #IMPLIED>
<!ATTLIST text BACKGROUND %colors; "WHITE">
<!ATTLIST text FOREGROUND %colors; "BLACK">

<!ELEMENT password ( (%events;)* | (%borders;)? )*>
<!ATTLIST password
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        VALUE CDATA #IMPLIED
        ECHOCHAR CDATA #IMPLIED>
<!ATTLIST password BACKGROUND %colors; "WHITE">
<!ATTLIST password FOREGROUND %colors; "BLACK">

<!ELEMENT textarea ( (%events;)* | font? | (%borders;)? )*>
<!ATTLIST textarea
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        VALUE CDATA #IMPLIED
        COLUMNS CDATA #IMPLIED
        ROWS CDATA #IMPLIED>
<!ATTLIST textarea BACKGROUND %colors; "WHITE">
<!ATTLIST textarea FOREGROUND %colors; "BLACK">

<!ELEMENT groupbutton ((checkbox|radiobutton|togglebutton)+ )?>
<!ATTLIST groupbutton
        COLUMNREF CDATA #IMPLIED
        ID CDATA #IMPLIED>

<!ELEMENT checkbox (( %events; )*|( %borders;)?|font? )*>
```

```
<!ATTLIST checkbox
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        CAPTION CDATA #IMPLIED>
<!ATTLIST checkbox SELECTED  %truefalse; "FALSE">
<!ATTLIST checkbox ENABLED  %truefalse; "TRUE">
<!ATTLIST checkbox PAINTEDBORDER %truefalse; "FALSE">
<!ATTLIST checkbox BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST checkbox FOREGROUND %colors; "BLACK">

<!ELEMENT radiobutton (( %events; )*|( %borders;)?|font? )*>
<!ATTLIST radiobutton
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        CAPTION CDATA #IMPLIED>
<!ATTLIST radiobutton SELECTED  %truefalse; "FALSE">
<!ATTLIST radiobutton ENABLED  %truefalse; "TRUE">
<!ATTLIST radiobutton PAINTEDBORDER %truefalse; "FALSE">
<!ATTLIST radiobutton BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST radiobutton FOREGROUND %colors; "BLACK">

<!ELEMENT togglebutton (( %events; )*|( %borders;)?|font? )*>
<!ATTLIST togglebutton
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        CAPTION CDATA #IMPLIED>
<!ATTLIST togglebutton SELECTED  %truefalse; "FALSE">
<!ATTLIST togglebutton ENABLED  %truefalse; "TRUE">
<!ATTLIST togglebutton BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST togglebutton FOREGROUND %colors; "BLACK">

<!ELEMENT button (( %events; )*|( %borders;)?|font?)*>
<!ATTLIST button
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        CAPTION CDATA #IMPLIED>
<!ATTLIST button SELECTED  %truefalse; "FALSE">
<!ATTLIST button ENABLED  %truefalse; "TRUE">
<!ATTLIST button BACKGROUND %colors; "LIGHTGRAY">
<!ATTLIST button FOREGROUND %colors; "BLACK">

<!ELEMENT listbox ( (%events;)*  | font? | (%borders;)?)*>
<!ATTLIST listbox
```

```
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        LOOKUPTABLE CDATA #IMPLIED
        LOOKUPCOLUMN CDATA #IMPLIED
        DISPLAYCOLUMN CDATA #IMPLIED
        WHERE CDATA #IMPLIED
        ORDERBY CDATA #IMPLIED
        VALUE CDATA #IMPLIED
        CODE CDATA #IMPLIED
        SELECTED CDATA #IMPLIED>
<!ATTLIST listbox BACKGROUND %colors;  "WHITE">
<!ATTLIST listbox FOREGROUND %colors; "BLACK">

<!ELEMENT combobox ( (%events;)*  | font? | (%borders;)?)*>
<!ATTLIST combobox
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        LOOKUPTABLE CDATA #IMPLIED
        LOOKUPCOLUMN CDATA #IMPLIED
        DISPLAYCOLUMN CDATA #IMPLIED
        WHERE CDATA #IMPLIED
        ORDERBY CDATA #IMPLIED
        VALUE CDATA #IMPLIED
        CODE CDATA #IMPLIED
        SELECTED CDATA #IMPLIED
        ROWS CDATA #IMPLIED>
<!ATTLIST combobox EDITABLE  %truefalse; "TRUE">
<!ATTLIST combobox BACKGROUND %colors;  "WHITE">
<!ATTLIST combobox FOREGROUND %colors; "BLACK">

<!ELEMENT grid ( (gridColumn+, gridData?)?, (%borders;)? )?>
<!ATTLIST grid
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        TABLEREF CDATA #IMPLIED
        WHERE CDATA #IMPLIED
        ORDERBY CDATA #IMPLIED>
<!ATTLIST grid SORTABLE  %truefalse; "TRUE">
<!ATTLIST grid COLRESIZABLE  %truefalse; "TRUE">
<!ATTLIST grid COLREORDER  %truefalse; "TRUE">
<!ATTLIST grid GRIDLINES  %truefalse; "TRUE">
<!ATTLIST grid BACKGROUND %colors;  "WHITE">
<!ATTLIST grid FOREGROUND %colors; "BLACK">
```

```
<!ELEMENT gridColumn ( (%events;)* | font?)? >
<!ATTLIST gridColumn
        ID CDATA #IMPLIED
        CAPTION CDATA #IMPLIED
        COLUMNREF CDATA #IMPLIED
        WIDTH CDATA #IMPLIED>
<!ATTLIST gridColumn EDITABLE  %truefalse; "TRUE">
<!ATTLIST gridColumn BACKGROUND %colors;  "WHITE">
<!ATTLIST gridColumn FOREGROUND %colors; "BLACK">

<!ELEMENT gridData EMPTY>
<!ATTLIST gridData
        ID CDATA #IMPLIED
        COLUMNSEPARATOR CDATA #IMPLIED
        ROWSEPARATOR CDATA #IMPLIED
        VALUE CDATA #IMPLIED>

<!ELEMENT slider ( (%events;)* | (%borders;)? )*>
<!ATTLIST slider
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        VALUE CDATA #IMPLIED
        MINIMUM CDATA #IMPLIED
        MAXIMUM CDATA #IMPLIED>

<!ELEMENT scrollbar ( (%events;)* | (%borders;)? )*>
<!ATTLIST scrollbar
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        VALUE CDATA #IMPLIED
        MINIMUM CDATA #IMPLIED
        MAXIMUM CDATA #IMPLIED>

<!ELEMENT progressbar ( (%events;)* | (%borders;)? )*>
<!ATTLIST progressbar
        ID CDATA #IMPLIED
        POS CDATA #REQUIRED
        COLUMNREF CDATA #IMPLIED
        VALUE CDATA #IMPLIED
        MAXIMUM CDATA #IMPLIED>

<!ELEMENT tree ((treefolder*,treeleaf*)|( %events; )* | (%borders;)? )*>
<!ATTLIST tree
        ID CDATA #IMPLIED
```

```
        POS CDATA #IMPLIED
        CAPTION CDATA #IMPLIED
        TABLEREF CDATA #IMPLIED
        IDCOLUMNREF CDATA #IMPLIED
        PARENTCOLUMNREF CDATA #IMPLIED
        DISPLAYCOLUMN CDATA #IMPLIED
        WHERE CDATA #IMPLIED
        ORDERBY CDATA #IMPLIED>
<!ATTLIST tree EDITABLE  %truefalse; "FALSE">
<!ATTLIST tree BACKGROUND %colors;  "WHITE">

<!ELEMENT treefolder ((treefolder*,treeleaf*)|( %events; )* )>
<!ATTLIST treefolder
        ID CDATA #IMPLIED
        CAPTION CDATA #REQUIRED>

<!ELEMENT treeleaf (( %events; )* )?>
<!ATTLIST treeleaf
        ID CDATA #IMPLIED
        CAPTION CDATA #REQUIRED>

<!ELEMENT colorchooser (%borders;)?>
<!ATTLIST colorchooser
        ID CDATA #IMPLIED
        POS CDATA #IMPLIED>

<!ELEMENT filechooser (%borders;)?>
<!ATTLIST filechooser
        ID CDATA #IMPLIED
        POS CDATA #IMPLIED>

<!ELEMENT directorypane (%borders;)?>
<!ATTLIST directorypane
        ID CDATA #IMPLIED
        POS CDATA #IMPLIED>

<!ELEMENT htmlpane  (%borders;)? >
<!ATTLIST htmlpane
        ID CDATA #IMPLIED
        POS CDATA #IMPLIED
        SOURCE CDATA #REQUIRED>

<!ELEMENT tabpane ((tab+) | (%borders;)? )?>
<!ATTLIST tabpane
        ID CDATA #IMPLIED
        POS CDATA #IMPLIED>
```

```
<!ELEMENT tab ( (%events;)*  | font? | (%borders;)? |
          panel? | (%controls;)* )*>
<!ATTLIST tab
      ID CDATA #IMPLIED
      TIP CDATA #IMPLIED
      CAPTION CDATA #IMPLIED>

<!ELEMENT messageBox EMPTY>
<!ATTLIST messageBox
      ID CDATA #IMPLIED
      TITLE CDATA #IMPLIED
      MESSAGE CDATA #REQUIRED>
<!ATTLIST messageBox TYPE %dialogTypes; "PLAIN">
<!ATTLIST messageBox INTERNAL  %truefalse; "FALSE">

<!ELEMENT confirmBox EMPTY>
<!ATTLIST confirmBox
      ID CDATA #IMPLIED
      TITLE CDATA #IMPLIED
      MESSAGE CDATA #REQUIRED>
<!ATTLIST confirmBox TYPE %dialogTypes; "PLAIN">
<!ATTLIST confirmBox OPTION %dialogOptions; "default">
<!ATTLIST confirmBox INTERNAL  %truefalse; "FALSE">

<!ELEMENT optionBox EMPTY>
<!ATTLIST optionBox
      ID CDATA #IMPLIED
      TITLE CDATA #IMPLIED
      MESSAGE CDATA #REQUIRED
      LISTSEPARATOR CDATA #IMPLIED
      LIST CDATA #REQUIRED
      VALUE CDATA #REQUIRED>
<!ATTLIST optionBox TYPE %dialogTypes; "PLAIN">
<!ATTLIST optionBox OPTION %dialogOptions; "default">
<!ATTLIST optionBox INTERNAL  %truefalse; "FALSE">

<!ELEMENT inputBox EMPTY>
<!ATTLIST inputBox
      ID CDATA #IMPLIED
      TITLE CDATA #IMPLIED
      MESSAGE CDATA #REQUIRED
      LISTSEPARATOR CDATA #IMPLIED
      LIST CDATA #IMPLIED
      VALUE CDATA #IMPLIED>
<!ATTLIST inputBox TYPE %dialogTypes; "PLAIN">
```

```
<!ATTLIST inputBox INTERNAL  %truefalse; "FALSE">

<!ELEMENT font EMPTY>
<!ATTLIST font
        ID CDATA #IMPLIED
        NAME CDATA #IMPLIED
        SIZE CDATA #IMPLIED>
<!ATTLIST font STYLE %fontstyle; "PLAIN">

<!ELEMENT onAction  (script)+>
<!ATTLIST onAction
        ID CDATA #IMPLIED
        EVENT (PERFORMED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onAdjustment  (script)+>
<!ATTLIST onAdjustment
        ID CDATA #IMPLIED
        EVENT (VALUE_CHANGED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onComponent  (script)+>
<!ATTLIST onComponent
        ID CDATA #IMPLIED
        EVENT (HIDDEN | MOVED | RESIZED | SHOWN) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onContainer  (script)+>
<!ATTLIST onContainer
        ID CDATA #IMPLIED
        EVENT (ADDED | REMOVED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onFocus  (script)+>
<!ATTLIST onFocus
        ID CDATA #IMPLIED
        EVENT (GAINED | LOST) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onItem  (script)+>
<!ATTLIST onItem
```

```
        ID CDATA #IMPLIED
        EVENT (STATE_CHANGED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onKey  (script)+>
<!ATTLIST onKey
        ID CDATA #IMPLIED
        EVENT (PRESSED | RELEASED | TYPED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onMouse (script)+>
<!ATTLIST onMouse
        ID CDATA #IMPLIED
        EVENT (CLICKED | ENTERED | EXITED |
            PRESSED | RELEASED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onMouseMotion  (script)+>
<!ATTLIST onMouseMotion
        ID CDATA #IMPLIED
        EVENT (DRAGGED | MOVED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onText  (script)+>
<!ATTLIST onText
        ID CDATA #IMPLIED
        EVENT (VALUE_CHANGED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onWindow  (script)+>
<!ATTLIST onWindow
        ID CDATA #IMPLIED
        EVENT (ACTIVATED | CLOSED | CLOSING | DEACTIVATED |
            DEICONIFIED | ICONIFIED | OPENED) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT onCode  (script)+>
<!ATTLIST onCode
        ID CDATA #IMPLIED
```

```
        EVENT (XMLDOCSTART | XMLDOCEND | SQLSTATEMENT |
DBINSERT |
            DBUPDATE | DBDELETE | DBSELECT | DBNEXT |
            DBPREVIOUS | FORMCLEAR | FORMDEFAULT ) #IMPLIED
        LANGUAGE %languages; #IMPLIED
        TARGET CDATA #IMPLIED>

<!ELEMENT script (#PCDATA)>
<!ATTLIST script
        ID CDATA #IMPLIED>

<!ELEMENT emptyBorder EMPTY>
<!ATTLIST emptyBorder
        ID CDATA #IMPLIED
        THICKNESS CDATA #IMPLIED>

<!ELEMENT lineBorder EMPTY>
<!ATTLIST lineBorder
        ID CDATA #IMPLIED
        THICKNESS CDATA #IMPLIED>
<!ATTLIST lineBorder COLOR %colors; "BLACK">

<!ELEMENT doubleLineBorder EMPTY>
<!ATTLIST doubleLineBorder
        ID CDATA #IMPLIED>
<!ATTLIST doubleLineBorder COLOR %colors; "BLACK">

<!ELEMENT etchedBorder EMPTY>
<!ATTLIST etchedBorder
        ID CDATA #IMPLIED>
<!ATTLIST etchedBorder HIGHLIGHT  %colors; #IMPLIED>
<!ATTLIST etchedBorder SHADOW  %colors; #IMPLIED>

<!ELEMENT bevelBorder EMPTY>
<!ATTLIST bevelBorder
        ID CDATA #IMPLIED
        HIGHLIGHT  %colors; #IMPLIED
        SHADOW  %colors; #IMPLIED
        HIGHLIGHTINNER  %colors; #IMPLIED
        HIGHLIGHTOUTER  %colors; #IMPLIED
        SHADOWINNER  %colors; #IMPLIED
        SHADOWOUTER  %colors; #IMPLIED>
<!ATTLIST bevelBorder SOFTCORNERS %truefalse; "FALSE">
<!ATTLIST bevelBorder TYPE (RAISED | LOWERED) "LOWERED">

<!ELEMENT titledBorder (font?)>
```

```
<!ATTLIST titledBorder ID CDATA #IMPLIED>
<!ATTLIST titledBorder TITLE CDATA #IMPLIED>
<!ATTLIST titledBorder COLOR  %colors; "BLACK">
<!ATTLIST titledBorder POSITION (ABOVETOP | TOP | BELOWTOP |
                ABOVEBOTTOM | BOTTOM | BELOWBOTTOM) "TOP">
<!ATTLIST titledBorder JUSTIFICATION (LEFT | CENTER | RIGHT) "LEFT">

<!ELEMENT compoundBorder ((%sborders;),(%sborders;))>
<!ATTLIST compoundBorder
      ID CDATA #IMPLIED>

<!ELEMENT sql ( Database )+ >

<!ELEMENT database ( table* | sqlstatement* )* >
<!ATTLIST database
      ID CDATA #IMPLIED
      NAME CDATA #REQUIRED
      USERID CDATA #IMPLIED
      PASSWORD CDATA #IMPLIED
      DRIVER CDATA #REQUIRED>

<!ELEMENT table ( indexes, columns ) >
<!ATTLIST table
      ID CDATA #IMPLIED
      NAME CDATA #REQUIRED
      INDEX CDATA #REQUIRED>

<!ELEMENT indexes ( index )+ >
<!ATTLIST indexes
      ID CDATA #IMPLIED>

<!ELEMENT columns ( column )+ >
<!ATTLIST columns
      ID CDATA #IMPLIED>

<!ELEMENT index ( columnRef )* >
<!ATTLIST index
      ID CDATA #IMPLIED
      NAME CDATA #REQUIRED
      TYPE (PRIMARY | ALTERNATE) "PRIMARY">
<!ATTLIST index DUPLICATE %truefalse; "NO">

<!ELEMENT column EMPTY>
<!ATTLIST column
      ID CDATA #IMPLIED
      NAME CDATA #REQUIRED
```

```
        TYPE (CHAR | VARCHAR | INT | FLOAT | DATE) #REQUIRED
        DEFAULT CDATA #IMPLIED
        LENGTH CDATA #IMPLIED>

<!ELEMENT columnRef EMPTY>
<!ATTLIST columnRef
        ID CDATA #IMPLIED
        NAME CDATA #REQUIRED>

<!ELEMENT sqlstatement EMPTY>
<!ATTLIST sqlstatement
        ID CDATA #IMPLIED
        STATEMENT CDATA #REQUIRED>
```